Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Comparação de Sequências Biológicas Utilizando Computação Heterogênea com OpenCL

Guilherme Rodrigues Costa

Brasília
2013

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Comparação de Sequências Biológicas Utilizando Computação Heterogênea com OpenCL

Guilherme Rodrigues Costa

Monografia apresentada como requisito parcial
para conclusão do Curso de Computação — Licenciatura

Orientadora
Prof.ª Dr.ª Alba Cristina Magalhães Alves de Melo

Brasília
2013

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Curso de Computação — Licenciatura

Coordenador: Prof. Dr. Flávio de Barros Vidal

Banca examinadora composta por:

Prof.ª Dr.ª Alba Cristina Magalhães Alves de Melo (Orientadora) — CIC/UnB
Prof.ª Dr.ª Carla Denise Castanho — CIC/UnB
Prof.ª Dr.ª Maria Emília M. T. Walter — CIC/UnB

CIP — Catalogação Internacional na Publicação

Costa, Guilherme Rodrigues.

Comparação de Sequências Biológicas Utilizando Computação Heterogênea com OpenCL / Guilherme Rodrigues Costa. Brasília : UnB, 2013.
60 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2013.

1. Computação heterogênea, 2. GPGPU, 3. OpenCL, 4. sequências biológicas, 5. bioinformática

CDU 004.4

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Comparação de Sequências Biológicas Utilizando Computação Heterogênea com OpenCL

Guilherme Rodrigues Costa

Monografia apresentada como requisito parcial
para conclusão do Curso de Computação — Licenciatura

Prof.ª Dr.ª Alba Cristina Magalhães Alves de Melo (Orientadora)
CIC/UnB

Prof.ª Dr.ª Carla Denise Castanho
CIC/UnB

Prof.ª Dr.ª Maria Emília M. T. Walter
CIC/UnB

Prof. Dr. Flávio de Barros Vidal
Coordenador do Curso de Computação — Licenciatura

Brasília, 5 de fevereiro de 2013

Agradecimentos

Agradeço primeiramente a Prof^a. Dr^a. Alba Cristina Melo por todo o apoio,confiança e empolgação apesar de todas as situações adversas que passamos no decorrer desse projeto. Gostaria de agradecer também a Prof^a. Dr^a. Maria Emília Machado Telles e Prof^a. Dr^a Carla Denise Castanho, por terem aceitado o convite de fazer parte da banca e dedicado seu tempo fazendo sugestões que muito contribuíram para a melhoria desse trabalho.

Agradeço a todos meus amigos da UnB que me ajudaram e acompanharam nessa jornada. Em especial agradeço a Cristiano *"Koshiro"*, Eduardo *"Kyrw"*, Felipe "Mestre", Luiggi *"Aeradon"* e Marcelo *"Marshall"* pelo apoio físico e psicológico nos meus momentos de dificuldades e pelas sessões de RPG que me tiraram muitas horas de estudo mas que me permitiram ganhar bons amigos.

Agradeço aos meus pais por todo o suporte e paciência, por me manterem acordado nas madrugadas e me ajudarem na luta para acordar pela manhã. Agradeço as minhas irmãs pelas revisões de textos e pela companhia nas madrugadas.

Resumo

O alinhamento de sequências possui uma diversidade de aplicações na bioinformática, sendo uma das mais importantes operações nessa área. Dentre os métodos de alinhamento temos os algoritmo de Smith-Waterman, que é um método baseado em programação dinâmica e que provê o melhor alinhamento entre duas sequências. Apesar de ser um método exato, ele é demorado devido a sua complexidade quadrática.

Para tentar reduzir esse tempo de execução desse método várias propostas foram feitas, dentre elas temos o uso de computação heterogênea. Computação heterogênea é o termo utilizado quando o ambiente computacional é composto de diferentes tipos de elementos de processamento.

O objetivo do presente Trabalho de Graduação é propor uma implementação do algoritmo de Smith-Waterman em OpenCL e testar sua utilização em CPU e GPU de forma a verificar a simplicidade de programação para múltiplos elementos de processamento e analisar possíveis ganhos em tempo de execução.

Palavras-chave: Computação heterogênea, GPGPU, OpenCL, sequências biológicas, bioinformática

Abstract

The sequence alignment has a wide range of applications in bioinformatics, one of the most important operations in that area. Among alignment methods there is the Smith-Waterman algorithm, which is a method based on dynamic programming that provides the best alignment between two sequences. Although it is an exact method, it is time consuming due to its quadratic complexity.

There are several proposals to try to reduce the execution period of this method, one of them is the use of heterogeneous computing. Heterogeneous computing is the strategy of deploying multiple types of processing elements within a single workflow, and allowing each to perform the tasks to which it is best suited.

The goal of this work is to propose an implementation of Smith-Waterman algorithm in OpenCL and test its use in CPU and GPU in order to verify the simplicity of programming to multiple processing elements and possible gains in runtime.

Keywords: Heterogeneous computing, GPGPU, OpenCL, biologic sequences, bioinformatics

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

O alinhamento de sequências possui uma diversidade de aplicações na bioinformática, sendo uma das mais importantes operações nessa área. Alinhamento de Sequências é o resultado do procedimento de comparação de duas ou mais sequências de DNA ou de proteínas para identificar regiões de similaridade [16]. Essas regiões podem conter informações funcionais, estruturais e evolucionárias sobre os organismos. Existem métodos exatos para comparação de sequências que são baseados em programação dinâmica [8] e possuem complexidade quadrática de tempo.

Os métodos baseados me programação dinâmica têm por objetivo gerar o alinhamento ótimo entre duas sequências. Incluem as semelhanças e diferenças entre os caracteres e lacunas dentro das sequências, de modo que resulte na maior pontuação possível. Foi provado matematicamente que esses métodos fornecem a melhor pontuação para o alinhamento entre duas sequências [16]. Pode-se buscar o alinhamento global ou alinhamento local. O foco desse trabalho está no alinhamento local, que é conhecido como algoritmo de Smith-Waterman [29].

O algoritmo de Smith-Waterman é bastante demorado devido a sua complexidade quadrática de tempo. Já foram propostos na literatura várias estratégias para redução do tempo de processamento desse algoritmo, como utilização de *cluster* [3] e GPU [7]. Apesar dos resultados obtidos com essas estratégias de execução do Smith-Waterman em plataformas de alto desempenho terem sido muito bons, a programação requer normalmente bastante tempo ou o uso de chamadas específicas do dispositivo (GPUs e multicores, por exemplo) que dificultam a portabilidade. A fim de utilizar vários elementos de processamento e, ainda assim, manter a portabilidade optamos por utilizar a OpenCL.

A OpenCL (*Open Computing Language*) [12] é uma arquitetura para escrever programas que funcionam em plataformas heterogêneas, consistindo em CPUs, GPUs e outras unidades de processamento. Além de permitir a programação multi-plataforma o OpenCL permite também a programação paralela usando tanto o paralelismo de tarefas como de dados. Dessa forma é possível escolher um dispositivo que seja mais adequado para uma determinada tarefa sem que haja, contudo, mudanças no código do programa.

O objetivo do presente Trabalho de Graduação é propor uma implementação do algoritmo de Smith-Waterman em OpenCL e testar sua utilização em CPU e GPU de forma a verificar a simplicidade de programação para múltiplos elementos de processamento e analisar possíveis ganhos em tempo de execução. Como estratégia adotou-se a múlti-

pla comparação de pares de sequências. Na UnB, esse é o primeiro trabalho que utiliza OpenCL.

O restante desse documento está organizado do seguinte modo. O Capítulo 2 discorre sobre comparação de sequências biológicas, definindo conceitos básicos sobre sequências biológicas e apresenta o algoritmo de Smith-Waterman. O Capítulo 3 explica o conceito de computação heterogênea e suas arquiteturas, mostrando também a história das GPUs e sua evolução. O Capítulo 4 apresenta o OpenCL e sua arquitetura. O Capítulo 5 apresenta a proposta de implementação do algoritmo Smith-Waterman em OpenCL e no Capítulo 6 são apresentados os resultados experimentais. Finalmente, o Capítulo 7 apresenta a conclusão e as sugestões de trabalhos futuros.

Capítulo 2

Comparação de Sequências Biológicas

Nesse capítulo é feita uma pequena introdução teórica sobre sequências biológicas e alinhamento de sequências. Além disso, é dado enfoque nos algoritmos de alinhamento de Needman-Wunsh e Smith-Waterman, que utilizam o método de programação dinâmica (*Dynamic Programming*).

2.1 Sequências Biológicas

Uma sequência biológica pode ser composta de DNA, RNA ou proteínas. O DNA ou ADN (ácido desoxirribonucleico) é um composto orgânico, em uma estrutura de dupla hélice [6], que contém informações sobre a estrutura, informações genéticas e funcionamento de um organismo. O DNA é formado por nucleotídeos que por sua vez consistem em três partes: fosfato, pentose (açúcar) e uma base nitrogenada [24]. As bases nitrogenadas encontradas no DNA são: adenina (A), citosina (C), guanina (G) e timina (T). Em um DNA cada base nitrogenada de uma fita é ligada a outra base correspondente na outra fita, formando um par de base. Esses pares de base são Adenina com Timina e Citosina com Guanina, conforme ilustrados na Figura 2.1.

O RNA ou ARN (ácido ribonucleico) também é um composto orgânico formado por nucleotídeos, porém possui uma estrutura de hélice simples. Além disso, no lugar da base nitrogenada timina, encontramos no RNA a base nitrogenada uracila (U), formando o par Adenina com Uracila.

Uma proteína é codificada por uma sequência de três bases nitrogenadas, por meio de uma estrutura de RNA chamada RNA mensageiro (RNAm). Assim, com os 4 pares de bases são gerados 64 tipos de códons, porém apenas 20 proteínas são codificadas. Na Tabela 2.1 são ilustrados os códons e as proteínas codificadas por eles.

2.2 Alinhamento de Sequências

Alinhamento de Sequências é o resultado do procedimento de comparação de duas ou mais sequências de DNA ou de proteínas para identificar regiões de similaridade [16]. O alinhamento de duas sequências é feito pela transcrição delas em duas linhas onde caracteres idênticos ou similares (*match*) são colocados na mesma coluna, e caracteres

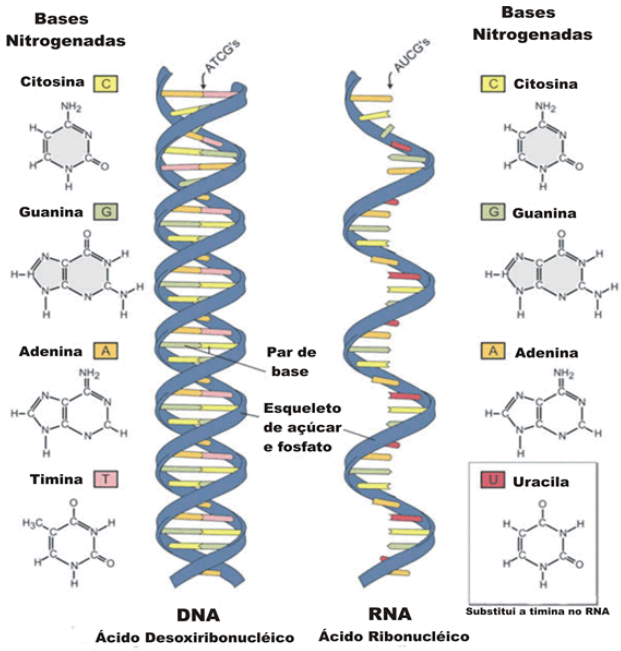

Figura 2.1: DNA e RNA: diferenças estruturais e suas bases nitrogenadas (Figura Adaptada de [20])

.

não idênticos podem ser colocados na mesma coluna (*mismatch*) ou uma lacuna (*gap*) pode ser introduzida.

Existem dois tipos de alinhamentos de pares de sequências: global e local [9]. No alinhamento global tenta-se alinhar a sequência inteira, utilizando todos os caracteres, incluindo ambas as extremidades de cada sequência. Para o alinhamento global, é apropriado que as sequências sejam bem similares e de tamanhos parecidos. No alinhamento local, subsequências das sequências originais são alinhadas, gerando, dessa forma, uma ou mais ilhas de combinações ou sub-alinhamentos nas sequências alinhadas [30]. Esse tipo de alinhamento é mais adequado para alinhamento de sequências que são semelhantes ao longo de algum trecho, mas diferentes em outros, sequências de tamanhos diferentes ou sequências que compartilham uma região ou domínio.

O alinhamento de sequências é usado para descobrir informações funcionais e evolucionárias em sequências de DNA ou proteínas. Para obter essas informações é interessante obter o melhor alinhamento possível (alinhamento ótimo). Obter o melhor alinhamento provê informações muito úteis no que diz respeito às relações das sequências, dando informação sobre quais caracteres na sequência devem estar na mesma coluna de alinhamento e quais foram as inserções em uma sequência e deleções na outra.

Sequências que são muito parecidas ou similares provavelmente têm a mesma função, seja ela de caráter regulatório, no caso de moléculas de DNA semelhantes, ou função bioquímica semelhante e uma estrutura tridimensional, como no caso de proteínas. Além disso, se a comparação de sequências de organismos diferentes possui alta similaridade, pode ser uma indicação de uma sequência ancestral em comum, e essas sequências são definidas como homólogas [16]. O alinhamento indica as mudanças que podem ter ocorrido

Tabela 2.1: Lista de Códons e as proteínas codificadas [11]

UUU	Phenylalanine	UCU	Serine	UAU	Tyrosine	UGU	Cysteine
UUC	Phenylalanine	UCC	Serine	UAC	Tyrosine	UGC	Cysteine
UUA	Leucine	UCA	Serine	UAA	Terminate	UGA	Terminate
UUG	Leucine	UCG	Serine	UAG	Terminate	UGG	Tryptophan
CUU	Leucine	CCU	Proline	CAU	Histidine	CGU	Arginine
CUC	Leucine	CCC	Proline	CAC	Histidine	CGC	Arginine
CUA	Leucine	CCA	Proline	CAA	Glutamine	CGA	Arginine
CUG	Leucine	CCG	Proline	CAG	Glutamine	CGG	Arginine
AUU	Isoleucine	ACU	Threonine	AAU	Asparagine	AGU	Serine
AUC	Isoleucine	ACC	Threonine	AAC	Asparagine	AGC	Serine
AUA	Isoleucine	ACA	Threonine	AAA	Lysine	AGA	Arginine
AUG	Methionine	ACG	Threonine	AAG	Lysine	AGG	Arginine
GUU	Valine	GCU	Alanine	GAU	Aspartic acid	GGU	Glycine
GUC	Valine	GCC	Alanine	GAC	Aspartic acid	GGC	Glycine
GUA	Valine	GCA	Alanine	GAA	Glutamic acid	GGA	Glycine
GUG	Valine	GCG	Alanine	GAG	Glutamic acid	GGG	Glycine

entre duas sequências homólogas e uma ancestral durante a evolução.

Com o início da análise de genomas e comparação de sequências em larga escala, tornou-se importante o reconhecimento de que a similaridade de sequências pode indicar vários tipos de relações de ancestralidade, ou até mesmo nenhuma. A teoria evolucionária dispõe de termos que são usados para descrever a relação entre sequências. Nesse caso, deve-se diferenciar homologia e similaridade. Homologia refere-se a semelhanças entre estruturas de diferentes organismos que possuem a mesma origem evolucionária. Por outro lado, similaridade refere-se à observação empírica das semelhanças entre duas sequências e que pode ser quantificada.

2.3 Métodos de Alinhamento de Sequências

2.3.1 Matriz de pontos (*Dot Matrix*)

A análise por *Dot Matrix* é uma forma simples de comparação entre duas sequências, que visa buscar possíveis alinhamentos entre elas [30]. Além disso, o método também é utilizado para encontrar repetições diretas e invertidas de sequências de proteínas e DNA, e predizer regiões no RNA que são complementares.

No *Dot Matrix*, toma-se duas sequências a serem analisadas. Uma sequência fica alinhada no topo da página e a outra é listada do lado esquerdo. Começando pelo primeiro caractere da segunda sequência, é feita a comparação com cada caractere da primeira sequência. Se os caracteres corresponderem, é marcado um ponto naquela coluna. O mesmo é feito com os demais caracteres da segunda sequência, até que toda a página tenha sido preenchida com pontos, que representam todos os emparelhamentos da primeira sequência com a segunda. Regiões com similaridades são distinguidas por uma linha diagonal de pontos. Pontos isolados fora de uma diagonal representam correspondências

eventuais que provavelmente não estão relacionadas a algum alinhamento significante. A Figura 2.2 apresenta um exemplo de *Dot Matrix*.

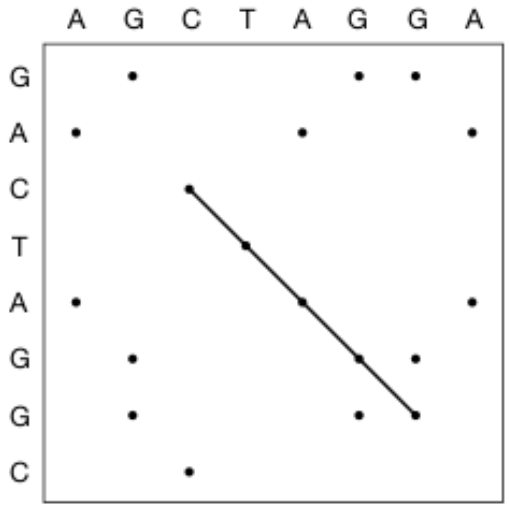

Figura 2.2: Simples comparação de sequências utilizando o Dot matrix [16].

A detecção de regiões de emparelhamento pode ser melhorada aplicando um filtro para remover as correspondências eventuais. Essa filtragem é atingida utilizando janelas (*sliding window*) para a comparação de duas sequências. Ao invés de comparar uma posição de um par de sequências são comparadas uma janela de posições adjacentes de duas sequências e um ponto só é marcado se uma certa quantidade de emparelhamentos (*stringency*) for atingida. Normalmente são usadas janelas maiores para sequências de DNA do que para sequências de proteínas porque o número de correspondências eventuais esperado entre sequências não relacionadas é maior devido ao fato de se usar apenas 4 símbolos para o DNA, comparado aos 20 símbolos para aminoácidos.

2.3.2 Programação Dinâmica (*Dynamic Programming*)

Programação Dinâmica é um método computacional bastante usado para alinhar duas sequências de proteínas ou de DNA. A Programação Dinâmica divide o problema em sub-problemas menores e os resolve na maneira dividir para conquistar [8]. Quando aplicada à comparação de sequências compara cada par de caracteres em duas sequências e gera um alinhamento. O alinhamento vai incluir as compatibilidades e incompatibilidades de caracteres e lacunas entre as sequências, de modo que o número de compatibilidades entre caracteres idênticos ou relacionados resulte na maior pontuação possível. O método é importante para análise de sequências, pois foi provado matematicamente que ele provê a melhor pontuação (alinhamento ótimo) para duas sequências em um dado sistema de pontuação.

O algoritmo de programação dinâmica tem por objetivo obter alinhamentos através da escolha de um sistema de pontuação (*scoring system*) para comparação de pares de caracteres e uma penalidade (*penalty score*) para lacunas. Idealmente, quando feito o alinhamento entre duas sequências de proteínas, por exemplo, espera-se encontrar longas

regiões com pares de aminoácidos idênticos ou similares e poucas lacunas. A qualidade do alinhamento entre duas sequências de proteínas é computado usando um sistema de pontuação que favorece o emparelhamento de aminoácidos relacionados ou idênticos e penalizando alinhamentos incompatíveis e lacunas. Para decisão sobre como devem ser pontuadas essas regiões, é necessário informações sobre os tipos de mudanças em sequências de proteínas relacionadas. Essas mudanças podem ser expressadas pela :

- probabilidade de se encontrar um par de aminoácidos em particular no alinhamento das sequências relacionadas;
- probabilidade de se alinhar o mesmo par de aminoácido por acaso nas sequências, dado que alguns aminoácidos são abundantes nas proteínas e outros raros;
- inserção de uma lacuna de um ou mais resíduos em uma das sequências, desse modo forçando que o alinhamento de cada par de aminoácido com outro aminoácido seja a melhor escolha.

Tanto o alinhamento global quanto o alinhamento local podem ser calculados com programação dinâmica. O alinhamento global é obtido com o algoritmo de Needleman-Wunsh [18] e o alinhamento local é obtido com o algoritmo de Smith-Waterman [29].

Alinhamento Global: Algoritmo de Needleman-Wunsh

A programação dinâmica garante obter um alinhamento ótimo ou a melhor pontuação entre duas sequências. Porém, o alinhamento encontrado continua a depender da escolha da pontuação para os pareamentos, incompatibilidades e lacunas. Existem dois tipos de alinhamento que podem ser computados: o alinhamento global, que inclui toda a sequência, e o alinhamento local que inclui somente as partes das sequência que produzem o alinhamentos com maiores pontuações. O algoritmo de programação dinâmica, com pequenas modificações, pode produzir tanto o alinhamento global quanto o alinhamento local. As diferenças entre o alinhamento global e o alinhamento local estão nos tipos de pontuação que são mantidos na matriz de pontuação da programação dinâmica. No algoritmo de Needleman-Wunsh, que produz o alinhamento global, todas as pontuações são mantidas, sejam elas positivas ou negativas. No algoritmo de Smith-Waterman, que produz o alinhamento local, apenas os valores positivos são mantidos, sendo que os valores negativos são convertidos em zero.

Para produzir um alinhamento global, tem-se duas fases a serem seguidas: o preenchimento da matriz (*matrix fill*) e a obtenção do melhor alinhamento global. Para produzir o melhor alinhamento global é calculada uma matriz de programação dinâmica H, que conterá em cada posição $H_{i,j}$, a pontuação do melhor alinhamento dos prefixos das sequências $a_{[1...i]}$ e $b_{[1...j]}$, bem como uma seta indicando a célula que foi usada para produzir essa pontuação. Para duas sequências $\mathbf{a} = a_1a_2...a_n$ e uma sequência $\mathbf{b} = b_1b_2...b_m$ a formulação matemática para o preenchimento da matriz H é calculada com a Equação 2.1.

$$H_{i,j} = \max \begin{cases} H_{i-1,j-1} + s(a_i b_j) \\ H_{i,j-1} - w_y \\ H_{i-1,j} - w_x \end{cases} \tag{2.1}$$

Onde $H_{i,j}$ é a pontuação na posição i da sequência **a** e posição j da sequência **b**, $s(a_i b_j)$ é a pontuação de alinhamento dos caracteres nas posições i e j, w_x é a penalidade por lacunas de tamanho x na sequência **a** e w_y é a penalidade por lacunas de tamanho y na sequência **b**.

Tomando por exemplo as sequências $ATTGCC$ e $ATGCAA$, e os valores 1 para emparelhamento, -1 para incompatibilidade e -2 para lacunas, a Figura 2.3 ilustra a matriz H.

	*	A	T	T	G	C	C
*	**0**	−2	−4	−6	−8	−10	−12
A	−2	**1**	−1	−3	−5	−7	−9
T	−4	−1	**2**	0	−2	−4	−6
G	−6	−3	0	**1**	**1**	−3	−5
C	−8	−5	−2	−1	0	**2**	0
A	−10	−7	−4	−3	−2	0	**1**
A	−12	−9	−6	−5	−4	−2	**-1**

Figura 2.3: Preenchimento de matriz segundo algoritmo de Needleman-Wunsh para as sequências **ATTGCC** e **ATGCAA**

Para conseguir o alinhamento global, entre duas sequências a e b, percorrer-se a matriz a partir da posição i,j procurando sempre a maior pontuação nas posições adjacentes. Caso a posição de maior valor for a da célula superior, deve-se incluir uma lacuna na sequência **a**. Caso seja na célula a esquerda, insere-se uma lacuna na sequência **b**. Caso seja na diagonal nenhuma lacuna deve ser adicionada, mas adiciona-se o pareamento de a_i com b_j. Com isso, da matriz da Figura 2.3 o seguinte alinhamento mostrado na Figura 2.4:

$$\begin{array}{cccccccc} A & T & T & G & C & - & C \\ A & T & - & G & C & A & A \end{array}$$

Figura 2.4: Alinhamento obtido pelo algoritmo de Needleman-Wunsh, no exemplo da Figura 2.3

Alinhamento Local: Algoritmo de Smith-Waterman

Uma pequena modificação no algoritmo de programação dinâmica permite criar um alinhamento local de sequências dando a maior pontuação local entre duas sequências [29].

Normalmente, alinhamentos locais são mais significativos do que emparelhamentos locais, pois eles identificam domínios locais de sequências conservadas que estão presentes em ambas sequências. O algoritmo de Smith-Waterman não é particularmente adequado para achar a maior pontuação quando as sequências incluem várias regiões que emparelham localmente, mas são separadas por regiões que tem um emparelhamento fraco. No caso de alinhamento de duas sequências genômicas que incluem regiões de emparelhamento e incompatibilidades, encontrar o alinhamento local exige um grande custo computacional. O algoritmo de Smith-Waterman garante encontrar as melhores regiões de emparelhamento para quaisquer pares de DNA ou sequência de proteínas.

As regras para calcular os valores na matriz de pontuação são um pouco diferentes das utilizadas no algoritmo de Needleman-Wunsh. As diferenças mais importantes são a inicialização da primeira linha e primeira linha e coluna com zero e, embora o sistema de pontuação inclua pontuações negativas para incompatibilidades, quando o valor na matriz de pontuação na programação dinâmica for negativo, aquele valor deve ser convertido a zero, que tem o efeito de terminar qualquer alinhamento além daquele ponto. Os alinhamentos são produzidos começando pelas posições de maior pontuação e traçar um caminho dessas posições para a posição de pontuação zero.

Para produzir os melhores alinhamento locais, assim como no alinhamento global, é calculada uma matriz de programação dinâmica H, que conterá em cada posição $H_{i,j}$, a pontuação do melhor alinhamento dos prefixos das sequências $a_{[1...i]}$ e $b_{[1...j]}$, bem como uma seta indicando a célula que foi usada para produzir essa pontuação. A formulação matemática para o algoritmo de programação dinâmica é revisto para incluir a escolha do zero como valor em qualquer posição da matriz. Para duas sequências $\mathbf{a} = a_1a_2...a_n$ e uma sequência $\mathbf{b} = b_1b_2...b_m$ a formulação matemática para o preenchimento da matriz H, segundo Smith e Waterman [29], é calculada com a equação 2.2.:

$$H_{i,j} = \max \begin{cases} H_{i-1,j-1} + s(a_i b_j) \\ H_{i,j-1} - w_y \\ H_{i-1,j} - w_x \\ 0 \end{cases} \tag{2.2}$$

Onde $H_{i,j}$ é a pontuação na posição i da sequência $\mathbf{a}$ e posição j da sequência $\mathbf{b}$, $s(a_ib_j)$ é a pontuação de alinhamento dos caracteres nas posições i e j, w_x é a penalidade por lacunas de tamanho x na sequência $\mathbf{a}$ e w_y é a penalidade por lacunas de tamanho y na sequência $\mathbf{b}$.

Tomando as sequências $ATTGCC$ e $ATGCAA$, e os valores 1 para emparelhamento, -1 para incompatibilidade e -2 para lacunas, a matriz *trace-back* para essas sequências deve ficar como mostrado na Figura 2.5.

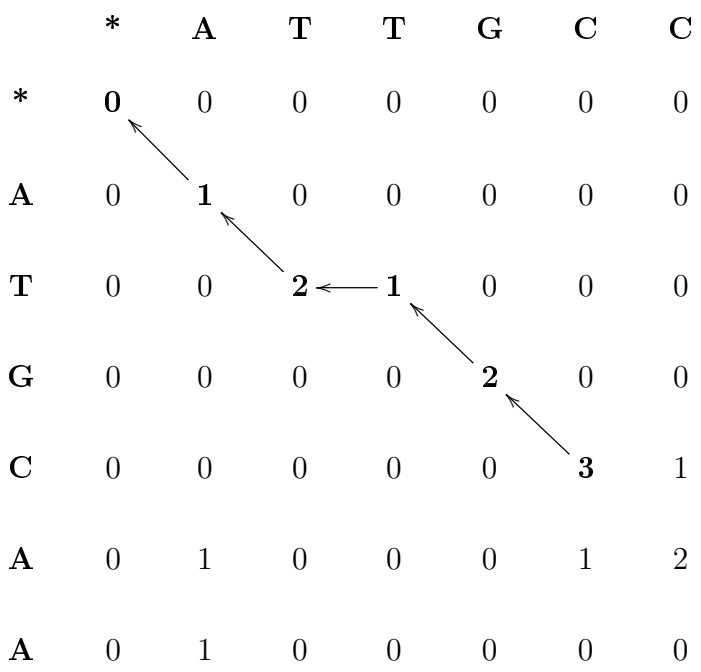

Figura 2.5: Preenchimento de matriz segundo algoritmo de Smith-Waterman para as sequências **ATTGCC** e **ATGCAA**, notando-se que o maior valor da matiz é 3, indicando uma região com boa similaridade local.

Assim, com a matriz já preenchida podemos realizar o alinhamento. Para conseguir o alinhamento local, começamos pela célula de maior valor, nesse caso a célula de valor 3, e percorremos a matriz até alcançar uma célula de valor zero. Assim, da matriz mostrada na Figura 2.6 obtemos o alinhamento mostrado na Figura 2.6

$$\begin{array}{cccccc} A & T & T & G & C \\ A & - & T & G & C \end{array}$$

Figura 2.6: Alinhamento obtido pelo algoritmo de Smith-Waterman

Capítulo 3

Computação Heterogênea

Computação heterogênea é o termo utilizado quando o ambiente computacional é composto de diferentes tipos de elementos de processamento [26]. Esse modelo pode empregar os processadores especializados para acelerar algumas operações, sendo mais rápido do que processadores escalares, ao expandir a aplicabilidade de arquiteturas de microprocessadores convencionais. Desse modo, o processamento heterogêneo permite que seja escolhido um tipo de processador "ótimo" [15] para cada operação dentro de uma determinada aplicação.

Atualmente, CPUs *multi-core* utilizam grande parte de seu processamento com lógica e *cache*, consumindo muita energia em unidades não-computacionais [4]. Arquiteturas heterogêneas oferecem uma alternativa a essa estratégia: arquiteturas *multicore* tradicionais em combinação com *cores* aceleradores. *Cores* aceleradores são designados para maximizar a performance dada uma potência fixa ou orçamento do transistor. Isso normalmente significa que os *cores* aceleradores utilizam menos transistores e executam em frequências mais baixas que as CPUs tradicionais. Funcionalidades complexas também são sacrificadas, desativando sua capacidade de executar sistemas operacionais, por exemplo. Os *cores* aceleradores são normalmente geridos por *cores* tradicionais para trabalhar com operações de uso intensivo de recursos. O tipo ótimo e a composição de processadores tradicionais e aceleradores, porém, varia de uma aplicação para outra [4].

3.1 *Hardware* e *Software* Tradicionais

Primeiramente, será usado o termo *chip* para designar um único circuito físico, e *core* para designar um núcleo físico de processamento, devido ao fato de o termo processador ter se tornado ambíguo, podendo se referir a um *chip*, um *core* ou um *core* virtual. Existem várias camadas de paralelismo nos *hardwares* atuais, inclusive as seguintes:

- **Paralelismo *Multi-chip*:** múltiplos *chips* físicos de processamento em um único computador de compartilhando recursos através do qual o custo de comunicação é relativamente barato.
- **Paralelismo Multi-core:** é similar ao paralelismo *multi-chip*, mas os núcleos dos processadores estão contidos em um único chip. Isso faz a comunicação ser ainda mais barata.

- **Paralelismo Multi-contexto (*thread*):** é visto quando um único núcleo pode mudar entre múltiplas execuções de contexto com pouco ou nenhum custo. Cada contexto requer um arquivo de registros e contador de programa em *hardware* separados.
- **Paralelismo de Instrução:** é quando um processador pode executar mais de uma instrução em paralelo, usando múltiplas unidades de instrução.

Atualmente *CPUs* utilizam várias dessas técnicas para diminuir os ciclos por instrução e mascarar a latência de memória. Exemplos incluem:

- ***Pipelining***: múltiplas instruções estão simultaneamente em *pipeline*.
- **Vetor de paralelismo**: uma instrução é replicada por toda a série de unidades aritméticas.
- **Processadores superescalares**: múltiplas instruções são enviadas para unidades de execução diferentes, automaticamente, em hardware, ou explicitamente usando *Very Long Instruction Words* (VLIWs).

Existem também várias técnicas que têm por objetivo a latência de memória. A execução *Out-of-order* reordena um fluxo de instruções para minimizar o número de paradas do processador causado por latência na dependência de dados. *Hardware multithreading* permite que um conjunto de contextos de execução compartilhe as mesmas unidades de execução. A *CPU* instantaneamente alterna entre esses contextos quando há paradas de requisição de memória, o que diminui o impacto da latência.

Tradicionalmente, operações com ponto flutuante eram consideradas onerosas, enquanto recuperar dados era praticamente ínfimo. Entretanto, essa situação mudou e o acesso a memória cresceu, a ponto de ser um fator limitante em muitas aplicações. Dados são movidos de dentro para fora do processador usando um número limitado de pinos a uma frequência limitada, conhecido como gargalo de von Neumann [2]. Além disso, existe uma certa latência para iniciar a transferência de dados.

De 1980 a 1996, a distância entre a performance do processador e da memória cresceu 50% [22]. Para reduzir esta disparidade, hierarquias de memória de grande porte têm sido desenvolvidas, abordando tanto o gargalo de von Neumann e a latência de memória quanto copiando os dados solicitados para memória mais rápida. Isto permite o acesso rápido aos dados usados recentemente, aumentando o desempenho para aplicações com acesso regular à memória . A seguir, são classificadas hierarquias de memória de acordo com a latência [4]:

- **Registradores**: são as unidades mais próximas de memória em um núcleo do processador e operaram na mesma velocidade que as unidades computacionais.
- **Memória Local**: se assemelha a um *cache* de dados programável com movimento explícito e tem uma latência de dezenas de ciclos de *clock*.
- **Memória Cache**: tem crescido rapidamente em tamanho e número. Em processadores modernos, normalmente há uma hierarquia de duas ou três camadas que operam com uma latência de dezenas de ciclos de *clock*. *Off-chip caches*, também encontrados em alguns computadores, tem uma latência em algum lugar entre *on-chip cache* e a memória principal.

- **Memória principal**: tem uma latência de centenas de ciclos de clock, e é limitada em largura de banda pelo o gargalo de von Neumann.

3.2 Arquiteturas Heterogêneas

A tendência de aumentar a performance através de paralelismo em vez de se aumentar a frequência de *clock*, tem se mostrado presente nas últimas gerações de processadores. Antes o foco estava no paralelismo em um único nó, em que a instrução em nível de paralelismo é quase totalmente explorada [13]. Isto significa que o aumento de desempenho deve vir de paralelismo por multi-chip, multi-core ou multi-contexto. Além disso outras estruturas de processadores têm se desenvolvido. A segunda geração de processadores da família Intel Core, mostrada na Figura 3.1 por exemplo, possui múltiplos *cores* além de um processador gráfico embutido no *chipset*.

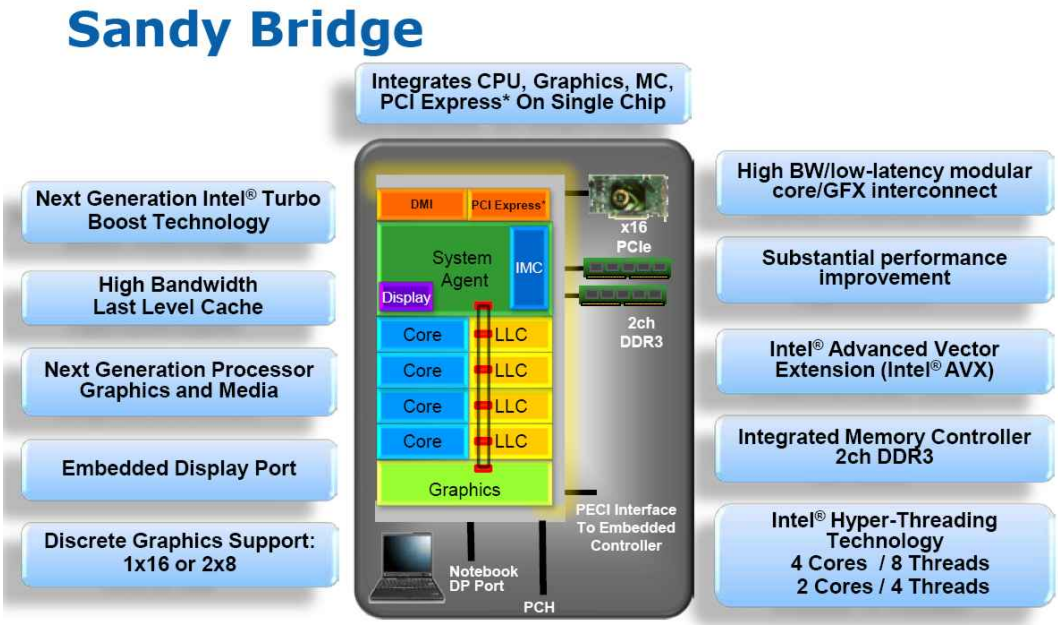

Figura 3.1: Microarquitetura Sandy Bridge, utilizada nos processadores da segunda geração da família Intel Core [5].

A taxonomia de Flynn [10] define quatro níveis de paralelismo em hardware:

- *Single Instruction Single Data* (SISD): corresponde à arquitetura dos dispositivos com um único processador, onde apenas uma instrução ou um fluxo de dados é processado a cada momento.
- *Single Instruction Multiple Data* (SIMD): mesma instrução executada simultaneamente sobre diversos conjuntos de dados.
- *Multiple Instruction Single Data* (MISD): constituída por uma *pipeline* de unidades de processamento independentes que operam sobre um mesmo fluxo de dados enviando os resultados de uma unidade para a próxima.

- *Multiple Instruction Multiple Data* (MIMD): Cada processador pode executar seu próprio fluxo de dados.

Além disso, duas subdivisões de MIMD são *Single Program Multiple Data* (SPMD) e *Multiple Program Multiple Data* (MPMD).

3.2.1 Breve Histórico das GPUs

A placa de vídeo é um dispositivo que permite a visualização de gráficos, figuras e outras mídias visuais utilizadas nos computadores. As placas de vídeos se desenvolveram juntamente com os computadores, quando eles começaram a usar telas para mostrar as informações necessárias [19].

A primeira placa de vídeo foi criada pela IBM em 1961 como forma de substituir os cartões impressos que eram usados em larga escala na época. A IBM lançou a primeira placa de vídeo incluída em um computador produzido em massa no modelo 8088, que foi o primeiro computador pessoal.

Adaptadores de vídeo monocromáticos (MDA) só funcionavam quando o computador estava no modo texto. Essas placas tinham 4MB de memória e resolução de 80 linhas por 25 colunas por tela. Além disso, apenas a a cor verde era disponível como cor da tela. Apenas no final dos anos 80, a cor vermelha foi acrescentada como cor da tela, mas a placa permitia mostrar apenas vermelho ou verde.

Os adaptadores gráficos de vídeo (VGA) se tornaram a nova geração de adaptadores de vídeo pois mostravam uma grande gama de cor e por isso eram mais aceitos que os monocromáticos. Devido a isso, muitas companhias como a *Cirrus Logic* e a ATI entraram no mercado quando componentes de computadores eram compatíveis entre si ainda que de fabricantes diferentes. Pouco tempo depois apareceu o Super VGA (SVGA) que possuía 256 cores e até 2MB de memória.

Em 1995, foram criadas as placas de 2D e 3D que tinham a capacidade de mostrar gráficos multidimensionais. Isso foi seguido dois anos mais tarde pela Voodoo [23], que era o mais poderoso chip gráfico da época. Juntos, a placa de vídeo e o processador aceleraram a capacidade dos desenvolvedores de software para criar gráficos mais realistas. Em 1997, a Intel projetou uma porta acelerada de gráficos (AGP) que permitia a conexão direta entre a placa gráfica e a memória com uma única porta na placa-mãe, acelerando a transferência de dados gráficos. Em 1999, a Nvidia tornou-se a fabricante dominante no mercado de placa de vídeo. Nesse momento a memória de vídeo 3D já possuíam de 32 MB até 125 MB.

Em 2001, a Nvidia apresentou a *GeForce* 3 que marcava um novo passo na evolução das GPUs: *pipeline* programável. Com essa tecnologia, em vez de mandar toda a descrição do gráfico para a GPU e seguir o fluxo do *pipeling* fixo, o programador podia mandar essa informação com os *vertex shaders* que operam sobre os dados, enquanto estiverem no pipeline.

Um ano depois, em 2002, as primeiras placas gráficas totalmente programáveis: Nvidia *GeForce* FX, ATI *Radeon* 9700. Essas placas permitiam operações por pixel com *vertex* programável e *pixel shaders*.

Em 2003, as primeiras placas de computação em GPU apareceram com a introdução do *DirectX* 9, tirando proveito da programabilidade agora no hardware GPU, mas não

para gráficos. Suporte total a valores de ponto flutuante a processamento avançado de textura começavam a aparecer nas placas de vídeo.

3.2.2 Evolução Histórica da GPU

Em decorrência da demanda de mercado por processamento em tempo real, de alta definição, de gráficos 3D, a GPU evoluiu para um processador altamente paralelizado, *multithread*, com vários núcleos, com grande poder computacional e com grande largura de banda, como ilustrada nas Figuras 3.2 e 3.3.

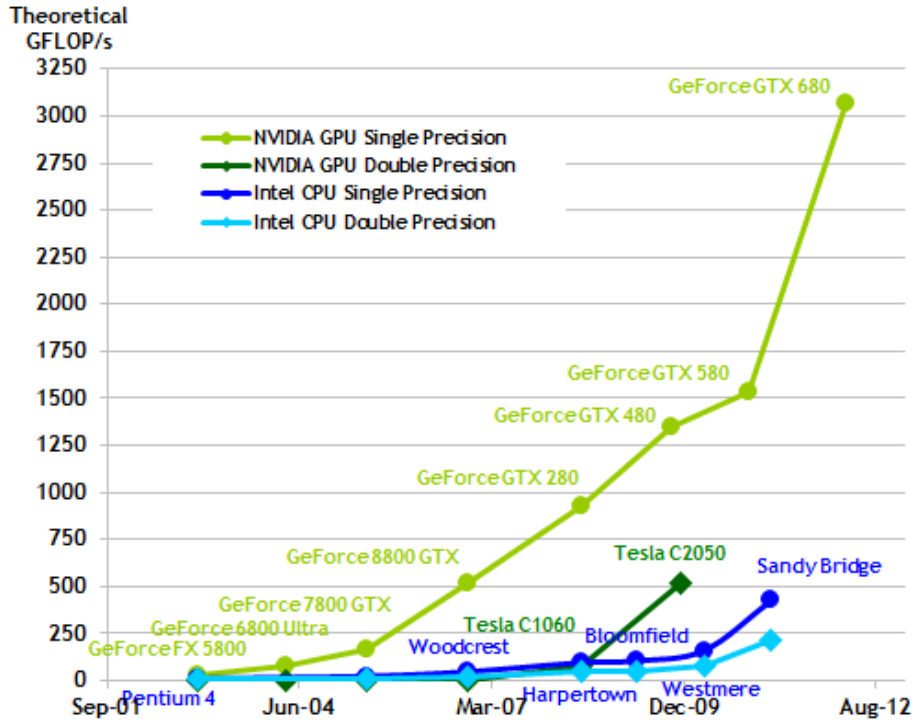

Figura 3.2: Operações de ponto flutuante por segundo ao longo do tempo [21].

A razão por trás da discrepância no poder computacional entre a CPU e a GPU é que a GPU é especializada em computação intensiva, altamente paralelizada - tendo em mente a renderização gráfica - portanto projetada para que mais transistores estejam dedicados ao processamento de dados do que para *data caching* e controle de fluxo, como esquematizado na Figura 3.4.

Mais especificamente, a GPU é especialmente adequada para resolver problemas que podem ser expressos como processamento de dados em paralelo - o mesmo programa é executado várias vezes para parcelas de dados diferentes, em paralelo com alta intensidade aritmética - relação de operações aritméticas comparadas com operações de memória. Como o mesmo programa é executado para cada porção de dados, a exigência por um sofisticado controle de fluxo é menor e, por ser executado repetidamente em um grande conjunto de dados e ter alta intensidade aritmética, a latência (demora) do acesso à memória pode ser escondida com cálculos em vez de caches com alta capacidade. Várias aplicações que processam grande conjuntos de dados podem usar o modelo de programação paralela de dados para aumentar a velocidade dos cálculos.

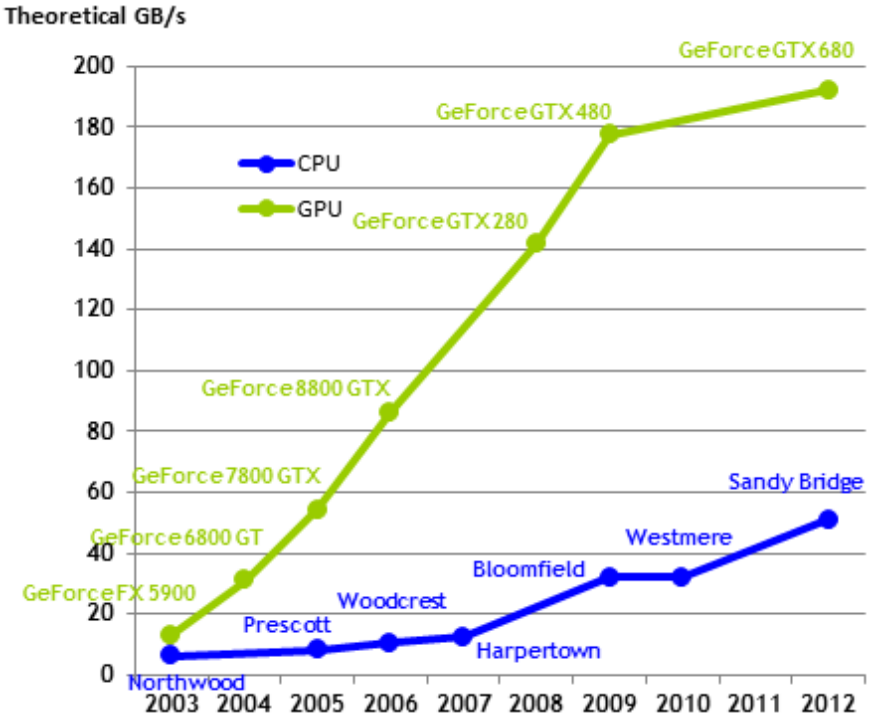

Figura 3.3: Largura de Banda da Memória para CPU e GPU ao longo do tempo [21].

Figura 3.4: Disposição de transistores para o processamento de dados em CPU e GPU [21]

Na renderização 3D, grandes conjuntos de *pixels* e vértices são mapeados para *threads* paralelas. Da mesma forma, aplicações de processamento de imagem e vídeo como pós-processamento de imagens renderizadas, codificação e decodificação de vídeo, redimensionamento de imagem visão estéreo, e reconhecimento de padrões podem mapear blocos de imagens e *pixels* para *threads* de processamento paralelo. Na realidade, vários algoritmos fora do campo de renderização de imagem e processamento são acelerados pelo processamento paralelo de dados, como processamento de sinais, simulação física, computação financeira ou biológica.

3.3 *General Purpose Graphical Process Unit* (GPGPU)

As unidades programáveis de GPU seguem o modelo de programação SPMD. Para aumentar a eficiência, a GPU processa muitos elementos (vértices ou fragmentos) em paralelo usando o mesmo programa. Cada elemento é independente dos outros elementos e, no modelo de programação base, elementos não podem se comunicar uns com os outros. Todos os programas GPU devem ser estruturados da seguinte forma: muitos elementos em paralelo, cada um processado em paralelo por um único programa.

Cada elemento pode operar em inteiros de 32-*bits* ou ponto flutuante com um conjunto de instruções razoavelmente completo de propósito geral. Elementos podem ler dados de uma memória global compartilhada (a operação *gather*) e, com as mais novas GPUs, também é possível rescrever os dados para locais arbitrários de memória global compartilhada (*scatter*).

Este modelo de programação é adequado para programas sequenciais, uma vez que muitos elementos podem ser processados em *lockstep* executando exatamente o mesmo código. O código escrito dessa maneira é SIMD. Como programas de *shader* se tornaram mais complexos, os programadores preferem permitir que diferentes elementos tomem caminhos diferentes através do mesmo programa, levando ao modelo SPMD mais geral.

Um dos benefícios da GPU é que grande parte de seus recursos são destinados à computação. Nas GPUs antigas, permitia-se caminhos diferentes de execução para cada elemento, sendo necessária uma quantidade substancial de hardware de controle. Hoje, GPUs contêm estruturas de controle de fluxo arbitrário por *thread*, porém impõem uma penalidade para a ramificação incoerente.

Nessas estruturas, elementos são agrupados em blocos, e blocos são processados em paralelo. Se elementos ramificam em diferentes direções dentro de um bloco, o hardware calcula ambos os lados do ramo para todos os elementos no bloco. O tamanho do bloco é conhecido como "granularidade de ramo" e vem diminuindo com as recentes gerações de GPU, sendo que atualmente é da ordem de 16 elementos.

Uma das dificuldades históricas na programação de aplicações GPGPU foi que, apesar de suas tarefas de uso geral normalmente não estarem relacionadas a aplicações gráficas, era obrigatória a programação utilizando APIs gráficas. Além disso, era necessário que o programa fosse estruturado em termos de *pipeline* gráfico, com as unidades programáveis acessíveis apenas como um passo intermediário em um *pipeline*, quando quase sempre é preferível acessar as unidades programáveis diretamente. Na atualidade, aplicações de computação GPU estão estruturadas da seguinte maneira:

- O programador define diretamente o domínio de computação de interesse como uma grade estruturada de *threads*.
- Um programa de propósito geral SPMD calcula o valor de cada *thread*.
- O valor para cada segmento é computado por uma combinação de operações matemáticas e acessos tanto de leitura (*gather*) quanto de escrita (*scatter*) a memória global. Ao contrário dos dois métodos anteriores, o mesmo *buffer* pode ser usado tanto para leitura quanto para escrita, permitindo algoritmos mais flexíveis.
- O *buffer* de memória global resultante pode então ser usado como entrada em computação no futuro.

Este modelo de programação é poderoso por várias razões. Primeiro, ele permite ao hardware explorar plenamente o paralelismo de dados do aplicativo especificando explicitamente aquele paralelismo no programa. Em seguida, estabelece um equilíbrio cuidadoso entre generalidade (uma rotina totalmente programável em cada elemento) e restrições para garantir um bom desempenho (o modelo SPMD, restrições à ramificação para a eficiência, as restrições à comunicação de dados entre os elementos e entre *kernels*, e assim por diante). Finalmente, seu acesso direto às unidades programáveis eliminam grande parte da complexidade enfrentada pelos programadores GPGPU anteriores ao optar pela utilização da interface gráfica para aplicações de uso geral de programação. Como resultado, os programas são mais frequentemente expressos em uma linguagem de programação familiar e são mais simples e mais fáceis de construir e depurar. O resultado é um modelo de programação que possibilita não só aproveitar ao máximo o poder de processamento da GPU, como também um modelo de programação de alto nível que incorpora a criação produtiva de aplicativos complexos.

Capítulo 4

OpenCL

A linguagem OpenCL (*Open Computing Language*) é uma API (*Application Programming Interface*) padrão para programação de computadores compostos de uma combinação de CPUs (*Central Processing Unit*), GPUs (*Graphics Processing Unit*) e outros processadores, como ilustrado na figura 4.1. Estruturas como essas são conhecidas como sistemas heterogêneos. A OpenCL é um padrão de indústria que permite a escrita de código multi-plataforma para execução nestes dispositivos, possibilitando a utilização de todo o poder computacional disponível no ambiente [1]. A OpenCL é baseada em um subconjunto estendido do padrão ISO C99 e, portanto, é muitas vezes referido como *OpenCL C*.

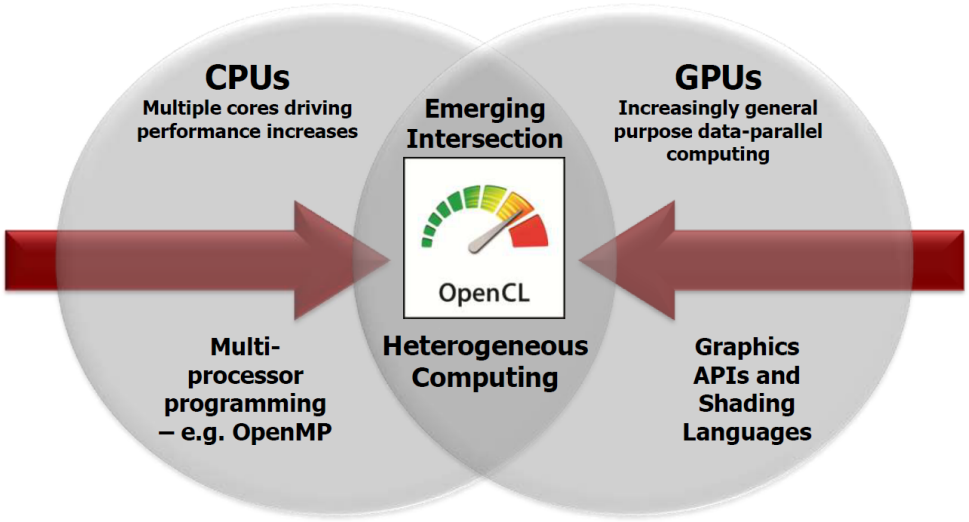

Figura 4.1: OpenCL possibilita criação de código multiplataforma, como para CPU e GPU [12].

4.1 Histórico

A OpenCL foi inicialmente desenvolvida pela Apple Inc., que detém os direitos autorais. Ela foi aprimorada numa proposta inicial de colaboração entre as equipes técnicas

da AMD, IBM, Intel e Nvidia. A Apple enviou sua proposta para o Khronos Group, um consórcio com foco na criação de padrões abertos livres. Assim, em 2008 foi criado o *Khronos Compute Working Group*, com representantes de empresas de CPU, GPU, processadores embarcados e software. A versão preliminar 1.0 da OpenCL foi lançada cinco meses depois, sendo revisada e lançada em Dezembro de 2008. A versão atual da OpenCL é a OpenCL 1.1.

4.2 Fundamentos de OpenCL

OpenCL suporta vários tipos de aplicações, porém as aplicações para plataforma heterogêneas devem seguir os seguintes passos [17]:

- Descobrir os componentes que compõem o sistema;
- Explorar as características desses componentes para que o software possa adaptar-se às características específicas dos diferentes elementos de hardware.
- Criar blocos de instrução (*kernels*) que irão ser executados na plataforma;
- Configurar e manipular os objetos de memória envolvidos;
- Executar os *kernels* na devida ordem assim como os componentes de sistema;
- Coletar os resultados.

Esses passos são atingidos a partir de uma série de APIs dentro da OpenCL e o ambiente de programação dos *kernels*. O fluxo de um programa OpenCL está representado na Figura 4.2. Dessa forma podemos dividir a programação nos seguintes modelos: Plataforma, Execução, Memória e Programação.

4.2.1 Modelo de Plataforma

O modelo de plataforma da OpenCL define representações de alto nível da plataforma heterogênea que será usada com a OpenCL. Esse modelo é mostrado na Figura 4.3. Cada plataforma OpenCL sempre inclui um único *host*, que pode ser uma CPU ou um dispositivo OpenCL.

O *host* interage com o ambiente externo através do programa OpenCL, incluindo interações de I/O (entrada/saída) com o usuário. Normalmente, a vazão do barramento interno do dispositivo é muito mais rápido do que a taxa de transferência do barramento externo entre o dispositivo e o *host*. Nesse caso a velocidade de um barramento externo não é um problema. Pelo fato da transferência de dados levar um longo tempo, é preciso fazer cálculos suficientes em cada *kernel* para ter certeza que não se está limitado por essa latência [14].

O *host* é conectado a um ou mais *OpenCL Devices*. O *device* é onde o fluxo de instruções ou *kernels* são executados. Um *device* pode ser uma CPU, GPU, DSP (*Digital Signal Processor*) ou qualquer outro tipo de hardware com suporte a OpenCL.

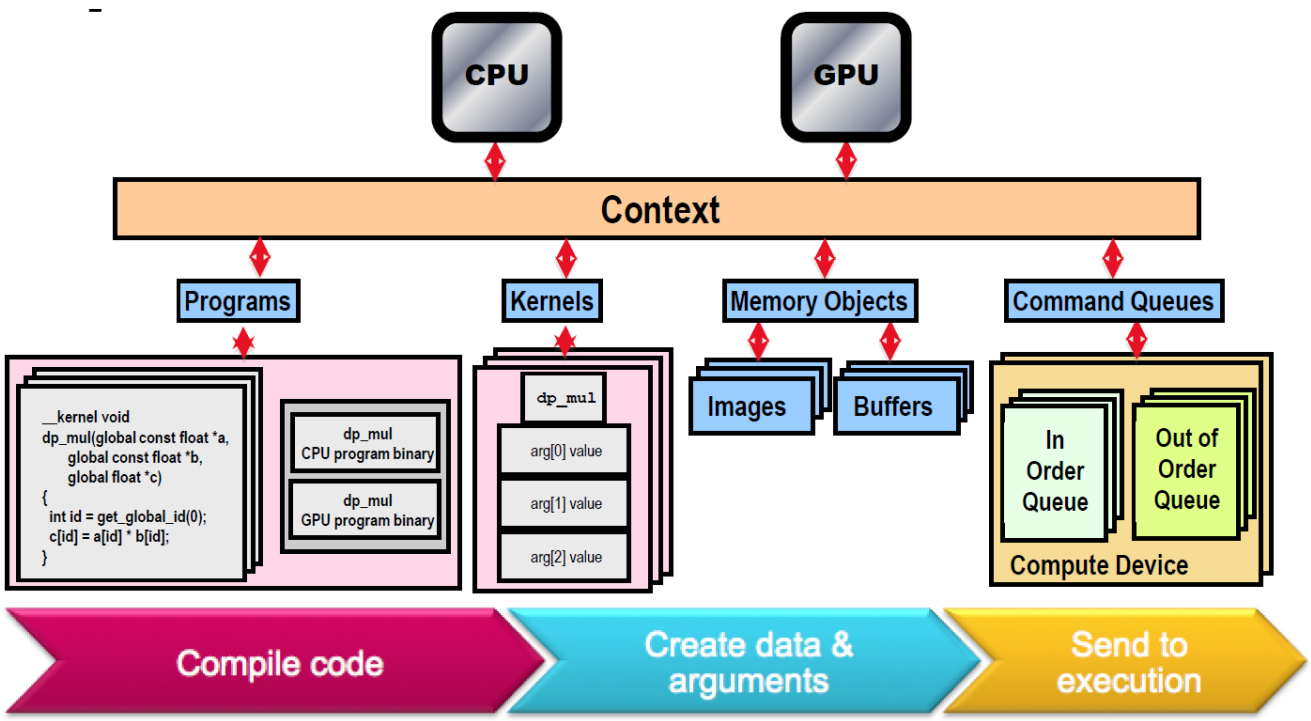

Figura 4.2: Fluxo de um programa em OpenCL. [12]

4.2.2 Modelo de Execução

Uma aplicação OpenCL consiste em duas partes distintas: o *host program* e uma coleção de um ou mais *kernels*. O *host program* é executado no *host*. A OpenCL não define detalhes como funciona o *host program*, apenas como ele interage com os objetos definidos na OpenCL. Tomemos por exemplo um problema de adição de dois vetores. No código descrito na Listagem 4.1 temos como ficaria um código para ser executado em uma CPU sem suporte a OpenCL.

```
1 void vector_add (const float* src_a,
2         const float* src_b,
3         float* res,
4         const int num)
5 {
6   for (int i = 0; i < num; i++)
7     res[i] = src_a[i] + src_b[i];
8 }
```

Listagem 4.1: Adição de dois Vetores em CPU

No entanto, para dispositivos OpenCL, o raciocínio seria um pouco diferente. Em vez de ter uma *thread* iterando sobre todos os elementos, poderíamos ter cada *thread* computando um elemento, sendo que cada *thread* corresponde a um elemento do vetor. Assim, o código ficaria como mostrado na Listagem 4.2.

```
1 __kernel void vector_add (__global const float* src_a,
2           __global const float* src_b,
```

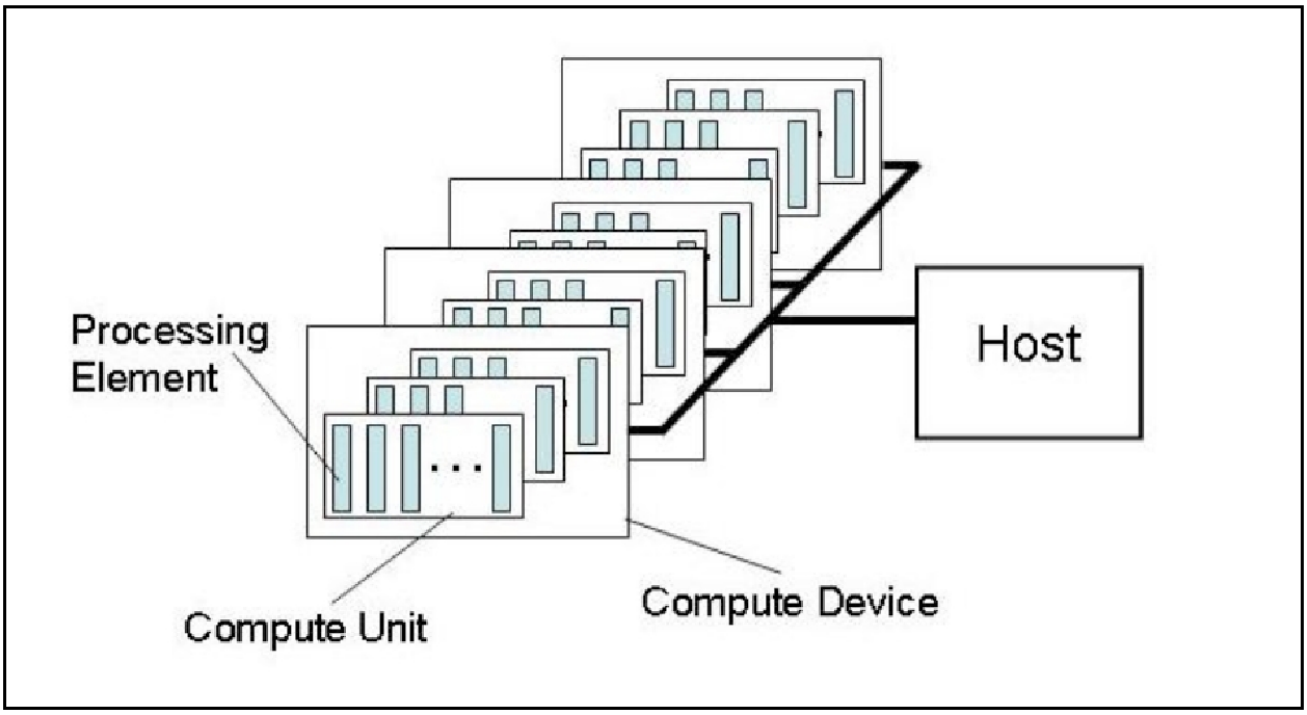

Figura 4.3: O modelo de plataforma da OpenCL com um *host* e outros dispositivos OpenCL. Cada dispositivo OpenCL tem uma ou mais unidades, cada qual com um ou mais elementos de processamento [17]

```
3             __global float* res,
4             const int num)
5 {
6    const int idx = get_global_id(0);
7
8    if (idx < num)
9       res[idx] = src_a[idx] + src_b[idx];
10 }
```

Listagem 4.2: Adição de dois Vetores em GPU

Os *kernels* são definidos no *host* e executados nos *OpenCL Devices*, e são eles que fazem todo o trabalho de uma aplicação OpenCL. *Kernels* são funções que transformam as entradas de memória em objetos de saída. A OpenCL define dois tipos de *kernels*:

- **OpenCL Kernels:** funções escritas com a linguagem OpenCL C e compiladas com o compilador OpenCL. Todas as implementações da OpenCL devem suportar *OpenCL Kernels*
- **Native Kernels:** funções criadas fora da OpenCL e acessada pela OpenCL através de ponteiro para função. Essas funções podem ser, por exemplo, definidas no código fonte do *host* ou exportada de uma biblioteca especializada.

A fim de generalizar a aplicação OpenCL para rodar em uma variedade de *hardwares*, é preciso definir no programa o número de *work-items* e *work-groups* a serem utilizados, já

que esses valores variam de acordo com o dispositivo e a necessidade do problema a ser resolvido. Usando a função `clGetDeviceInfo` com a *flag* `CL_DEVICE_MAX_WORK_ITEM_SIZES` e a função `clGetKernelWorkgroupInfo` com a *flag* `CL_KERNEL_WORK_GROUP_SIZE`, é possível determinar o tamanho máximo de grupo de trabalho para um determinado dispositivo e *kernel*. Este número muda de acordo com o dispositivo e *kernel*.

O aplicativo *host* configura o contexto em que os *kernels* irão executar, incluindo a alocação de memória de vários tipos (descritos na Seção 4.2.3), transferência de dados entre os objetos de memória e criação de *command-queues* que são usadas para controlar a sequência em que os comandos são executados, incluindo os comandos que executam *kernels*. O programador é responsável por sincronizar qualquer ordem de execução necessária.

O modelo de execução do OpenCL define como os *kernels* devem ser executados: em um dispositivo OpenCL ou num contexto.

Execução em um Dispositivo OpenCL

O *host program* emite um comando que submete o kernel para execução em um dispositivo OpenCL. Quando este comando é emitido pelo *host*, o *OpenCL runtime* cria um *index space* de inteiros. Uma instância do *kernel* executa em cada ponto no *index space*. Chamamos cada instância de um *kernel* em execução de *work item*, que é identificado pelas suas coordenadas no *index space*. Estas coordenadas são o ID global para o *work-item*. A estrutura do *space index* é mostrado na Figura 4.4.

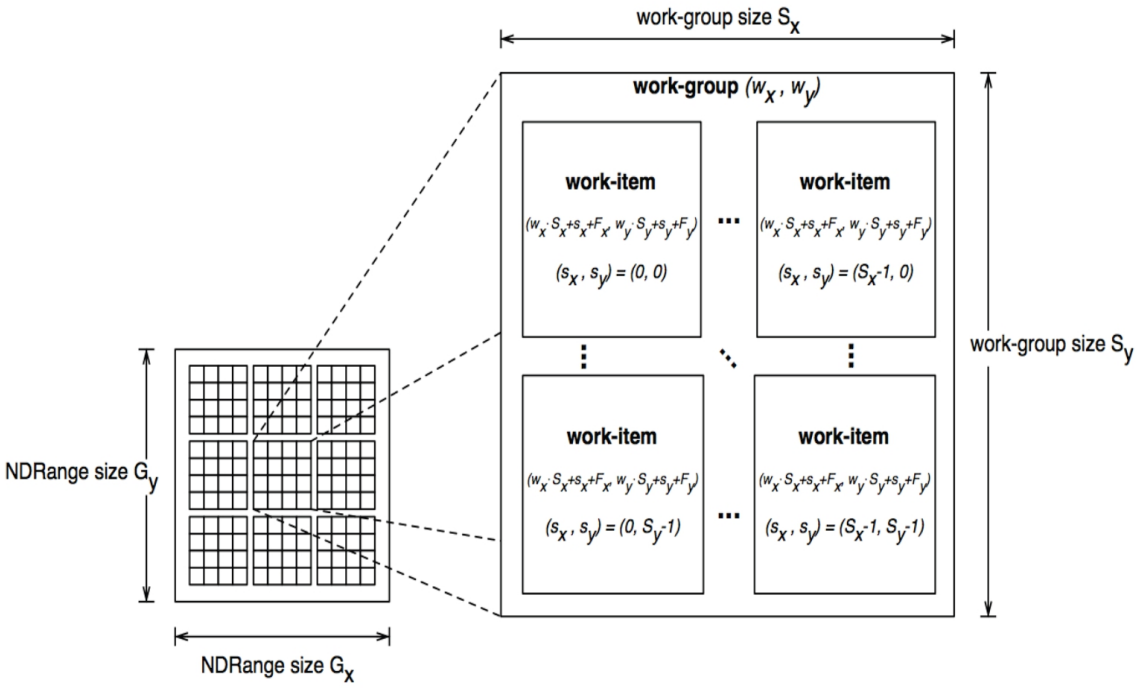

Figura 4.4: Execução de *kernels* - *Work-Groups* e *Work-Items* [25]

Work-Items são organizados em *work-groups*, assim como mostra a Figura 4.4. Os *work-groups* têm o mesmo tamanho em dimensões correspondentes, e este tamanho divide

uniformemente o *index space* em cada dimensão. A cada *work-group* é atribuído um ID único, com a mesma dimensionalidade como utilizado no *index space* para os *work-items*. Um ID único local dentro de um *work-group* é atribuído a cada *work-item* para que um único *work-item* possa ser identificado exclusivamente pelo seu ID global ou por uma combinação de seu ID local e ID de seu *work-group*. Os *work-groups* recebem IDs usando uma abordagem semelhante ao usado para *work-items*. Uma matriz de comprimento N define o número de *work-groups* em cada dimensão. *Work-items* são atribuídos a um *work-group* e é dado uma ID local com os componentes numa faixa de zero até o tamanho do *work-group* nessa dimensão menos um. Assim, a combinação da ID de *work-group* e da ID local dentro de um *work-group* define exclusivamente um *work-item*.

Os *work-items* em um dado *work-group* executam concorrentemente numa única unidade de processamento. Este é um ponto crítico no entendimento da concorrência em OpenCL. Uma implementação pode serializar a execução de *kernels*, podendo até serializar a execução de *work-groups* em uma única invocação de *kernel*. A OpenCL só garante que os *work-items* dentro de um *work-group* são executados concorrentemente (e dividem os recursos de processamento no dispositivo). Assim, não se pode assumir que *work-groups* ou invocações do kernel executam concorrentemente.

O *index space* abrange uma faixa de valores N-dimensionais, sendo assim chamada de *NDRange*. Atualmente o N pode assumir os valores 1, 2 ou 3. Em um programa OpenCL, um *NDRange* é definido por um *array* de inteiros de tamanho N especificando o tamanho do *index space* em cada dimensão. Cada ID global e local de *work-items* é uma tupla N-dimensional.

Execução em um Contexto

O trabalho computacional de uma aplicação OpenCL ocorre nos dispositivos OpenCL. O *host*, no entanto, desempenha um papel muito importante numa aplicação OpenCL. É no *host* onde os *kernels* são definidos. O *host* estabelece o contexto para os *kernels*. O *host* define o *NDRange* e as filas que controlam os detalhes de como e quando os *kernels* irão executar. Todas essas funções estão contidas na APIs dentro da OpenCL.

A primeira tarefa para o *host* é definir o contexto para a aplicação OpenCL. Como o nome indica, o contexto define o ambiente no qual os *kernels* são definidos e executados, como mostrado na Figura 4.5. O contexto pode ser definido em termos dos seguintes recursos [27]:

- ***Devices*:** coleção de *OpenCL Devices* para serem usadas pelo *host*.
- ***Kernels*:** funções OpenCL para serem executadas nos *OpenCL Devices*.
- ***Program Objects*:** código fonte do programa e executáveis que implementam os *kernels*.
- ***Memory Objects*:** conjunto de objetos em memória que são visíveis para os *OpenCL Devices* e contém valores que podem ser operados por instâncias de um *kernel*.

O contexto é criado e manipulado pelo *host* usando funções da API do OpenCL. Tomando como exemplo a estrutura da figura 4.6, como temos duas CPUs, o *host program* estará executando em uma delas. O *host program* irá requisitar ao sistema para listar

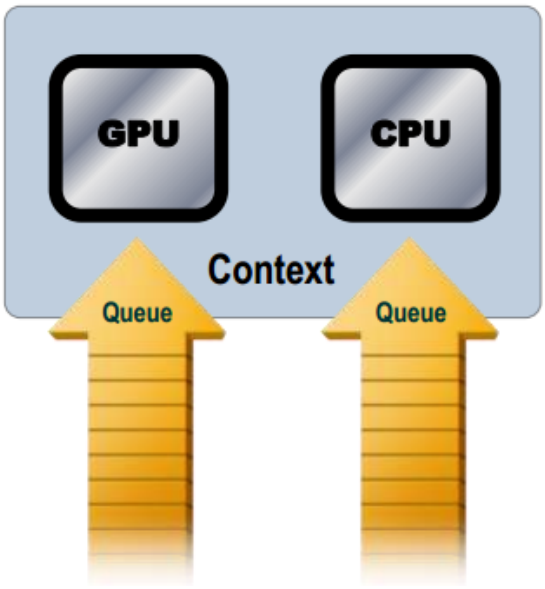

Figura 4.5: Exemplo de um contexto criado com uma CPU e uma GPU, tendo cada dispositivo uma *command_ queue* associada. [12]

quais são os dispositivos disponíveis, nesse caso a CPU e a GPU, e escolher quais dispositivos usar na aplicação OpenCL. Dependendo do tipo do problema e dos *kernels* a serem executados, o *host* pode escolher a GPU, outra CPU, outros *cores* na mesma CPU ou uma combinação deles. Uma vez feita esta escolha, define os dispositivos OpenCL no contexto atual.

Além disso, incluído no contexto, existe um ou mais *program objects* que contém o código para os *kernels*, que pode ser tanto um arquivo a ser carregado ou uma *string* definida no *host program* ou gerada dinamicamente dentro do *host program*.

A interação entre o *host* e os *OpenCL Devices* ocorre através de comandos emitidos pelo *host* para a *command-queue*. Esses comandos ficam em espera na *command-queue* até que sejam executados pelo *OpenCL Device*. Uma *command-queue* é criada no *host* e anexada a um *OpenCL Device* após o contexto ter sido definido. O *host* insere os comandos nas *command-queue*, e os comando são agendados para execução nos dispositivos associados. A *OpenCL* suporta três tipos de comandos [27]:

- **Comandos de execução de *kernel*:** executam um *kernel* nos elementos de processamento do *OpenCL Device*.
- **Comandos de memória:** transferem dados entre o *host* e diferentes objetos de memória, move dados entre objetos de memória, mapeia e libera objetos de memória do espaço de memória do *host*.
- **Comandos de sincronização:** inserem restrições quanto a ordem na qual os comandos devem ser executados.

Em um típico *host program*, são definidos o contexto e as *command-queues*, objetos de memória e de programa, e são construídos quaisquer tipos de estruturas de dados

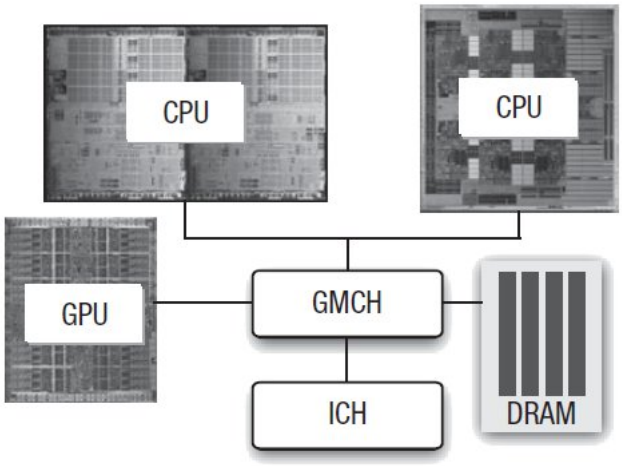

Figura 4.6: Uma plataforma heterogênea com dois soquetes, cada um com uma CPU multicore potencialmente diferente, um controlador de gráfico / memória (GMCH) que se conecta à memória do sistema (DRAM), e uma unidade de processamento gráfico (GPU). [17]

necessários no *host* para suportar a aplicação. Focando nas *command-queues*, objetos de memória são movidos do *host* para os dispositivos; argumentos do *kernel* são ligados a objetos de memória e, então, submetidos a *command-queue* para execução. Quando o *kernel* completa a execução, os objetos de memória produzidos durante a execução devem ser copiados de volta para o *host*.

Quando múltiplos *kernels* são submetidos para a fila, eles necessitam interagir. Um exemplo seria quando um conjunto de *kernels* geram objetos de memória que o próximo conjunto de *kernels* precisa manipular. Nesse caso, comandos de sincronização devem ser usados para forçar que o primeiro conjunto de *kernels* esteja completo antes da execução do conjunto seguinte.

Os comandos sempre executam assincronamente em relação ao *host program*. O *host program* envia comandos para a *command-queue* e então prossegue sem esperar que os comando finalizem. Se for necessário para o *host* aguardar por algum comando, isso deve ser explicitado com um comando de sincronização. Comandos dentro de uma única fila executam em relação uns aos outros em um dos dois modos:

- **Execução em ordem:** Comandos são executados na ordem na qual se encontram na *command-queue* e terminam nessa ordem, ou seja, um comando que entrou antes na fila deve ser completado antes do seguinte começar. Isso serializa a ordem de execução de comandos na fila.
- **Execução *Out-of-Order***: Comandos são executados na ordem mas não aguardam a conclusão dos anteriores para executar os próximos comandos. Qualquer restrição na ordem devem ser aplicada explicitamente, usando mecanismos de sincronização.

Para suportar protocolos de sincronização customizados, comandos emitidos para a *command-queue* geram objetos de evento. Um comando pode ser feito de modo a esperar por certas condições nos objetos de evento. Esses eventos podem ser usados também

para coordenar a execução entre o *host* e os dispositivos OpenCL. Além disso, é possível associar múltiplas filas a um único contexto para cada dispositivo OpenCL dentro daquele contexto. Essas duas filas ocorrem concorrentemente e independentemente, sem mecanismos explícitos da OpenCL para a sincronização entre elas. O código descrito em A.1 utiliza o *Wrapper* de C++ para OpenCL e exemplifica como deve ser estruturado um programa para ser executado no *host*, seguindo desde a criação de um contexto até a leitura dos dados da execução do *kernel*.

4.2.3 Modelo de Memória

A OpenCL define dois tipos de objeto de memória: *buffer objects* e *image objects*. Um *buffer object* é um bloco de memória disponível para os *kernels*. É possível mapear estruturas de dados para este *buffer* e acessá-lo através de ponteiros. Por outro lado, *image objects* são restritos a imagens. Um formato de imagem pode ser otimizado de acordo com a necessidade de um dispositivo OpenCL específico.

O OpenCL também permite que sejam especificadas sub-regiões de memória como objeto de memória distintos (incluído a partir da especificação da OpenCL 1.1 [27]). Com isso, uma sub-região de um grande objeto de memória como um objeto de primeira classe na OpenCL pode ser manipulado e coordenado através de uma *command-queue*.

O modelo de memória da OpenCL define cinco tipos distintos de região de memória [17]:

- ***Host Memory*:** essa região de memória é somente visível para o *host*. Assim como a maioria dos detalhes relativos ao host, a OpenCL define apenas como a *host memory* interage com *Opencl Objects* e *constructs*.
- ***Global Memory*:** essa região de memória permite acesso de leitura/escrita a todos os *work-items* em todos os *work-groups*. *Work-itens* podem ler de e escrever para qualquer elemento de um objeto de memória na memória global. Leituras e Escritas para a memória global podem ser armazenadas em *cache* dependendo dos recursos do dispositivo.
- ***Constant Memory*:** essa região de memória global permanece constante durante a execução de um *kernel*. O *host* aloca e inicializa os objetos de memória localizados na *constant memory*. *Work-Items* têm acesso somente leitura a esse objetos.
- ***Local Memory*:** essa região de memória é local ao *work-group* e pode ser usada para alocar variáveis que são compartilhadas a todos os *work-items* de um *work-group*. Ela pode ser implementada como regiões de memoria no dispositivo OpenCL. Alternativamente, a região da memória local pode ser mapeada em seções da memória global.
- ***Private Memory*:** essa região de memória é privada a um *work-item*. Variáveis definidas na memória privada de um *work-item* não visível a outros *work-items*.

Os *work-itens* são executados em elementos de processamento e tem suas próprias memórias provadas. Um *work-group* é executado em uma unidade de computação e compartilha uma região de memória local com os *work-items* no seu grupo. A memória

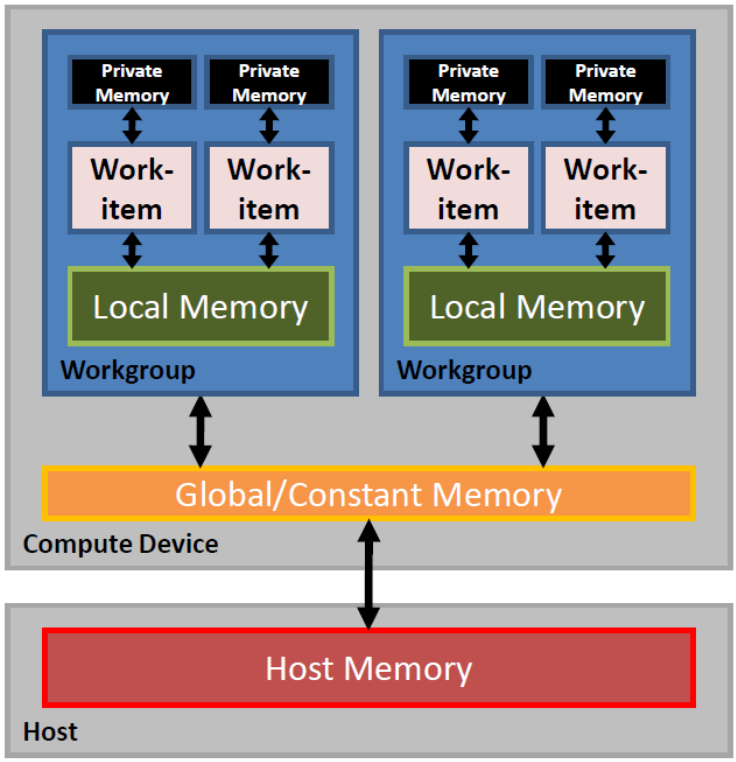

Figura 4.7: Modelo de memória da OpenCL [15].

do dispositivo OpenCL trabalha com o *host* para dar suporte à memória global como mostrado na Figura 4.7.

O modelo de memória do *host* e dispositivo OpenCL são, na maior parte das vezes, independentes um do outro. Isto ocorre por necessidade, uma vez que o *host* é definido fora da OpenCL. Eles, no entanto, às vezes precisam interagir. Esta interação ocorre de duas maneiras: copiando dados explicitamente ou através do mapeamento e desmapeamento de regiões de um objeto de memória.

Para copiar dados explicitamente, o *host* enfileira comandos para transferir dados entre o objeto de memória e a memória do *host*. Esses comandos de transferência de dados podem ser *blocking* ou *non-blocking*. Uma chamada de função OpenCL para uma transferência de memória bloqueada retorna uma vez que os recursos de memória associada no hospedeiro podem ser reutilizados com segurança. Para uma transferência de memória *non-blocking*, a chamada de função OpenCL retorna assim que o comando é enfileirado, independentemente se memória do *host* esteja segura para uso.

O método de mapeamento/desmapeamento de interação entre o *host* e objetos de memória OpenCL permite ao *host* mapear uma região do objeto memória em seu próprio espaço de endereço. O comando de mapeamento de memória (que é enfileirado na *command-queue* como qualquer outro comando OpenCL) pode ser *blocking* ou *non-blocking*. Uma vez que uma região de um objeto de memória é mapeado, o *host* pode ler ou escrever nessa região. O *host* desmapeia a região quando o *host* completa o acesso a essa região mapeada.

Quando a execução concorrente está envolvida, no entanto, o modelo de memória

precisa definir como os objetos da memória interagem com o *kernel* e o *host*. Este é o problema de consistência de memória. Não é suficiente dizer onde os valores de memória devem ir, mas deve-se também definir quando esses valores são visíveis a toda a plataforma. Dentro de uma memória de um *work-item* tem consistência de carga/armazenagem em um único *work-group* até o limite do *work-group*. Memória global é consistente em *work-items* em um único *work-group* até o limite do *work-group*, mas não há garantias da consistência de memória entre diferentes *work-items* executando um *kernel* [27].

4.2.4 Modelo de Programação

O modelo de execução OpenCL suporta os modelos de programação de dados paralelos (*Data-Parallel*) e de tarefas paralelas (*Task-Parallel*) , bem como suporta um híbrido desses dois modelos. O modelo principal que conduz o projeto OpenCL é o *Data-Parallel*.

O modelo de programação *Data-Parallel* define um cálculo em termos de uma sequência de instruções aplicadas a vários elementos de um objeto de memória. O *index space* associado a um modelo de execução define os *work-items* e como os dados devem ser mapeados para os *work-items*. Em um modelo estrito de programação paralela de dados, existe um mapeamento de um-para-um do *work-item* e o elemento em um objeto de memória através do qual um *kernel* pode ser executado em paralelo. A OpenCL define um modelo reduzido de um modelo de programação *Data-Parallel* onde o modelo estrito de mapeamento um-para-um não é requerido.

A Opencl provê paralelismo de dados hierárquico: paralelismo de dados dos *work-items* dentro de um *work-group* acrescido de paralelismo de dados ao nível dos *work-groups*. A especificação da OpenCL discute duas variantes dessa forma de paralelismo de dados. No modelo explícito, a responsabilidade de explicitar a definição dos tamanho dos *work-groups* recai sobre o programador. No segundo modelo, o modelo implícito, o programador apenas define o *NDRange space* e deixa a cargo do sistema a escolha dos *work-groups*.

Se o *kernel* não contém qualquer instrução condicional, cada *work-item* irá executar operações idênticas, mas em um subconjunto de itens de dados selecionados pelo seu *ID* global. Nesse caso, é importante definir um subconjunto do modelo de dados paralelos conhecido como *Single Instruction Multiple Data* (SIMD). Declarações condicionais dentro de um *kernel*, no entanto, podem levar cada *work-item* a executar operações muito diferentes. Embora cada *work-item* esteja usando o mesmo *kernel*, o resultado pode ser bastante diferente. Isso é muitas vezes conhecido como um modelo *Single Program Multiple Data* (SPMD).

O modelo de programação paralela de tarefas da OpenCL define um modelo no qual uma única instância de um *kernel* é executada independente de qualquer *space index*. É logicamente equivalente a executar um *kernel* em uma unidade de computação com um *work-group* contendo um único *work-item*.

Capítulo 5

Smith-Waterman em OpenCL

Nesse capítulo descreveremos uma forma de implementar o algoritmo de Smith-Waterman em OpenCL de modo a utilizar tanto a CPU quanto a GPU.

5.1 Visão Geral

Foi adotada como estratégia do projeto a múltipla comparação de pares de sequências biológicas, cabendo ao usuário escolher a melhor comparação de acordo com a pontuação máxima atingida na matriz. Por exemplo, se formos comparar 10 sequências entre si, serão feitas ao todo 45 comparações par-a-par. Como resultado, o usuário terá acesso a matriz de comparação e a pontuação máxima da cada matriz.

Para a execução desse projeto foi utilizada a *API OpenCL* de modo a criar um programa que utilize o processamento da *GPU* , aproveitando a estrutura de paralelismo que esse tipo de dispositivo fornece. Além disso, a escolha de *API OpenCL* foi feita pelo fato de ser possível selecionar outros dispositivos que tenham suporte a *OpenCL*, como a *CPU* com o mínimo de configuração e quase, ou nenhuma, alteração no código do kernel.

Para a execução do programa foi considerado um *index space* de 2 dimensões. Supondo que serão comparadas N sequências, cada dimensão terá um tamanho de *N-1*. Cada *Work-group* corresponde a uma sequência base e seus *Work-itens* correspondem à sequência de comparação. Os *Work-itens* têm sua identificação feita por um par ordenado (X,Y), sendo que a primeira coordenada corresponde à *ID* do seu *Work-group* e a segunda coordenada à sequência que deve ser comparada. Como exemplo e levando em consideração que o *ID* começa em zero, caso o *Work-item* tenha seu *ID*=(3,4), isso significa que terá como base a sequência 4 e que deverá ser feita a comparação com a sequência 5.

No final do processo, para um número N de sequências teremos $\frac{N(N-1)}{2}$ matrizes de pontuação, pois não são consideradas as comparações entre sequências do mesmo índice (Ex. ID=(2,2)) e comparações com índices simétricos já executados. Por exemplo, a comparação (4,3) não será executada pois a comparação (3,4) será feita primeiro.

Para reduzir a comunicação com o *Host*, optou-se por fazer uma única chamada de *kernel*, que executará todas as comparações. A interação entre o *Host* e o dispositivo é apresentado na Figura 5.1.

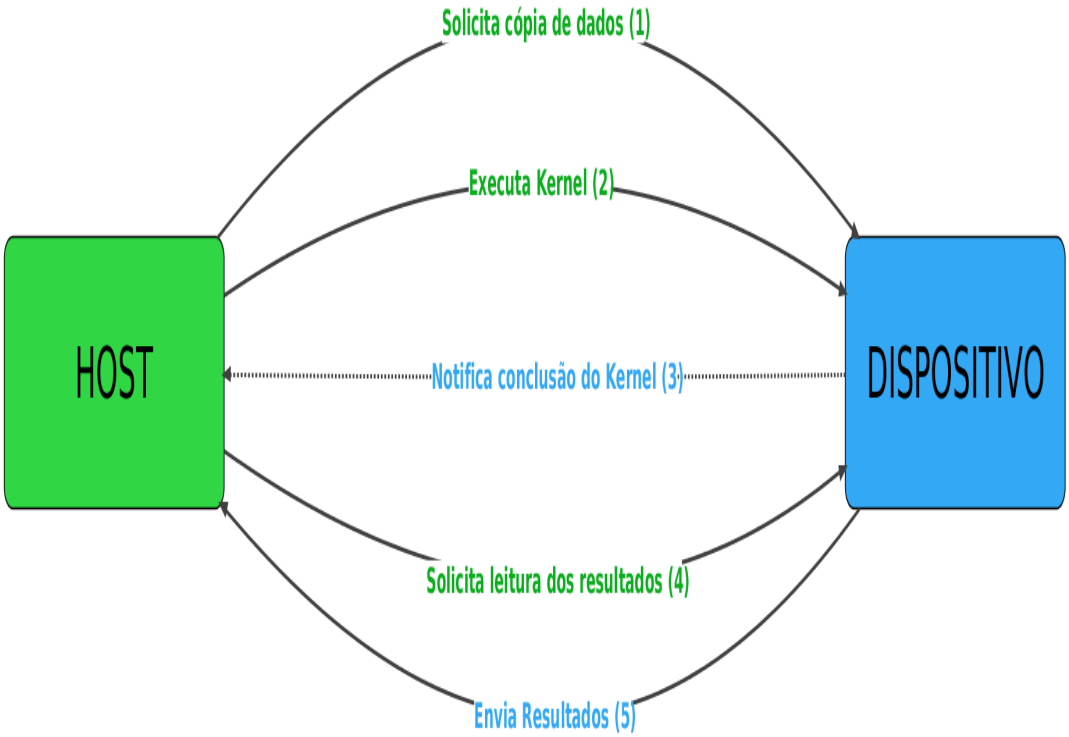

Figura 5.1: Interação entre *Host* e dispositivo.

Inicialmente, o *host* copia os dados de entrada para o dispositivo usando variáveis do tipo `cl::Buffer` usando a chamada `cl::Buffer(&context, flags, size, *host_ptr, *err)`, onde:

- *context* é a referência para o contexto criado.
- *flags* é um conjunto de opções sobre o tipo de *Buffer* a ser criado.
- *size* é o tamanho do *Buffer* a ser alocado.
- *host* é o ponteiro da variável do *Host* a ser copiada para o *Buffer*.
- *err* variável para receber o status da operação.

Tendo criado os *Buffers*, é preciso relacioná-los ao *kernel* usando o método `setArg(index, value )`, onde *index* é a posição do parâmetro na chamada da função e *value* a variável a ser alocada, que pode ser tanto *Buffer* quanto qualquer outra variável do programa. A próxima interação com o dispositivo é quanto o *Host* solicita a execução do *kernel* pelo comando `enqueueNDRangeKernel( &kernel, &offset,&global, &local,*events,event)`

- *kernel* é o objeto do *kernel* criado.
- *offset* é marca a partir de qual índice deve começar a identificação dos *work-itens*.
- *global* determina o tamanho das dimensões dos *work-groups*.

- *local* determina quantos *work-itens* devem ser definidor por *work-group*.
- *events* é uma lista de eventos pela qual o *kernel* fica esperando a sinalização para começar sua execução.
- *event* é o ponteiro para sinalizar os eventos da execução do *kernel*.

Ao finalizar a execução, o *kernel* sinaliza seu término para o *host*, através da variável de evento. O *host*, então, solicita a cópia dos dados do *buffer* de saída através do comando `enqueueReadBuffer(output_buf, blocking_read, offset , size_buf, hst_output,&events,&event)`, onde:

- *output_ buf* é o buffer de saída do dispositivo.
- *blocking_ read* marca se a leitura deve bloquear o fluxo do programa.
- *offset* marca a partir de qual índice deve ser feita a leitura.
- *size_ buf* deterimna o tamnho do *buffer* a ser lido.
- *hst_ output* é o ponteiro para a memória do *host* onde deve ser gravado o *buffer* do dispositivo.
- *events* é uma lista de eventos pela qual o comando deve esperar a finalização para iniciar sua execução.
- *event* é o ponteiro para sinalizar os eventos da leitura do *buffer*.

O fluxo de execução do programa projetado nesse trabalho de conclusão de curso é mostrado na Figura 5.2. Os módulos contidos nessa figura serão explicados na Seção 5.2, a seguir.

5.2 Descrição dos Módulos

5.2.1 Passagem de Parâmetros

A leitura das sequências é feita a partir de arquivos definidos no código do programa. Após serem definidos, as sequências são lidas e carregadas para a memória do *host*, sendo que cada linha do arquivo corresponde a uma sequência. Além disso, são passados como parâmetros de entrada o tipo de dispositivo a ser utilizado (como descrito na especificação da OpenCL 1.1 [27])), os números da plataforma e do dispositivo a ser utilizado, a penalidade de inserção e de remoção, nome do arquivo contendo o código do *kernel* e nome do *kernel* a ser utilizado. Todos esses argumentos são passados ao método *Setup()* que executa as rotinas necessárias do programa. Os argumentos solicitados são:

- *device_ type*: tipo de dispositivo a ser buscado.
- *platform*: índice da plataforma a ser usada.
- *device*: índice do dispositivo a ser utilizado.
- *sequences* : vetor com o endereço das sequências a serem comparadas.

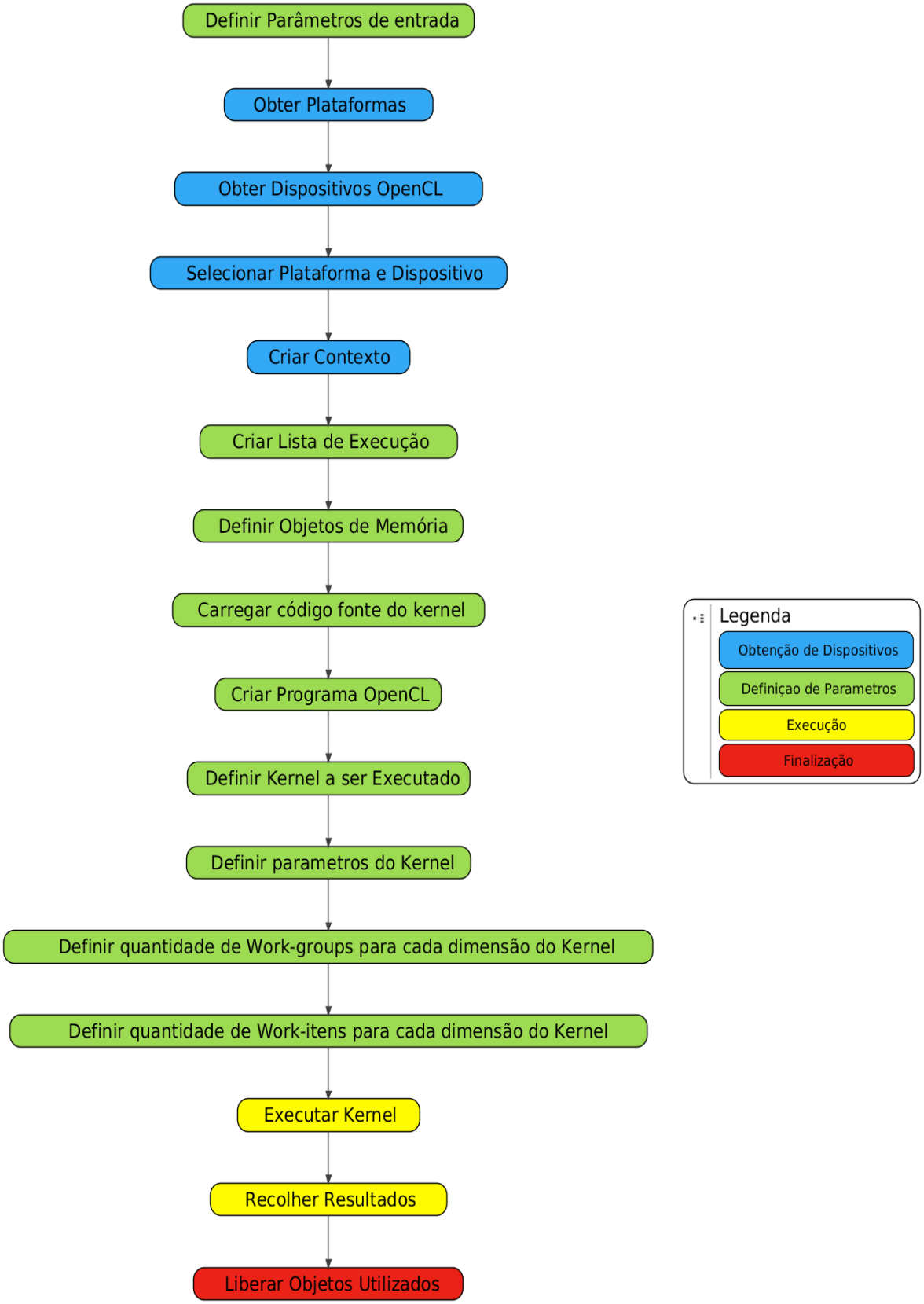

Figura 5.2: Fluxo de execução do programa usando a API OpenCL.

- *input_fee* : penalização para *gaps* na primeira sequencia.
- *remove_fee* : penalização para *gaps* na segunda sequência.
- *score_fee* : pontuação por emparelhamento.
- *kernel_file* : endereço para o arquivo que contém o *kernel*.
- *kernel* : nome do kernel a ser utilizado.
- *verbose* : define se serão mostradas mais informações durante a execução.

5.2.2 Escolha de Plataforma, Dispositivo e Contexto

Depois de ter passado os argumentos de entrada, dentro do método *Setup()* é chamado um método para buscar e relacionar as plataformas disponíveis no sistema. Utilizando o Wrapper C++ do OpenCL [28], é chamada a instrução `cl::Platform::get(&platforms)` que armazena em *plataforms* a lista das plataformas disponíveis. Caso exista alguma plataforma OpenCL disponível, são mostradas algumas informações da plataforma: nome, fabricante e versão da implementação da *OpenCL* usada.

Após terem sido obtidas as plataformas é feito a escolha da plataforma e do dispositivo a serem usados para ser criado o contexto. Através no número da plataforma é criado o contexto pelo comando `cl::Context(type,cps)`, onde *type* é o tipo de dispositivo a ser utilizado e *cps* uma tripla com propriedades definidas para o contexto. Com o contexto definido, é possível obter os dispositivos associados a esse contexto pelo comando `getInfo<CL_CONTEXT_DEVICES>` e através do número do dispositivo escolhido é guardado o ponteiro de referência para o mesmo.

5.2.3 Criacão da *Command-Queue* e Definição de Objetos de Memória

Em seguida, é criado a *command-queue* utilizando o contexto e a referência do dispositivo. O próximo passo na execução é definir os objetos de memória. Como a API OpenCL não permite a referência de objetos do tipo ponteiro para ponteiro e *arrays* de tamanhos variáveis, foi optado por utilizar um vetor para armazenar as sequências a serem utilizadas. Além disso, um outro vetor foi utilizado, que será chamado de vetor de tamanho, de forma a identificar o término de uma sequência dentro do vetor de sequências. Com o vetor de tamanho é possível identificar o início e o fim de uma sequência, bem como obter o tamanho desta, bastando fazer uma subtração. Pelo mesmo motivo dos vetores de sequência e de tamanho, o objeto de saída foi definido como vetor. Esse vetor de saída vai armazenar todas as matrizes de pontuação.

Para facilitar a indexação dessas matrizes, o vetor de saída tem tamanho definido por $(max\_seq\_size)*(nSeq)$, onde max_seq_size é o tamanho da maior sequência e $nSeq$ é o número de sequências a serem comparadas. Para que o dispositivo *OpenCL* tenha acesso a esses objetos, são criados *buffers* apontando para esses objetos. Na definição desses *buffers* são utilizadas *flags* para determinar o tipo de acesso daquele *buffer*, ou seja entra e saída, e como deve ser esse acesso. Para uma melhor performance, foram utilizadas nos *buffers* de entrada as *flags* `CL_MEM_ALLOC_HOST_PTR` e `CL_MEM_COPY_HOST_PTR`. A primeira *flag* indica que seja alocado um espaço de memória no dispositivo *OpenCL* e a

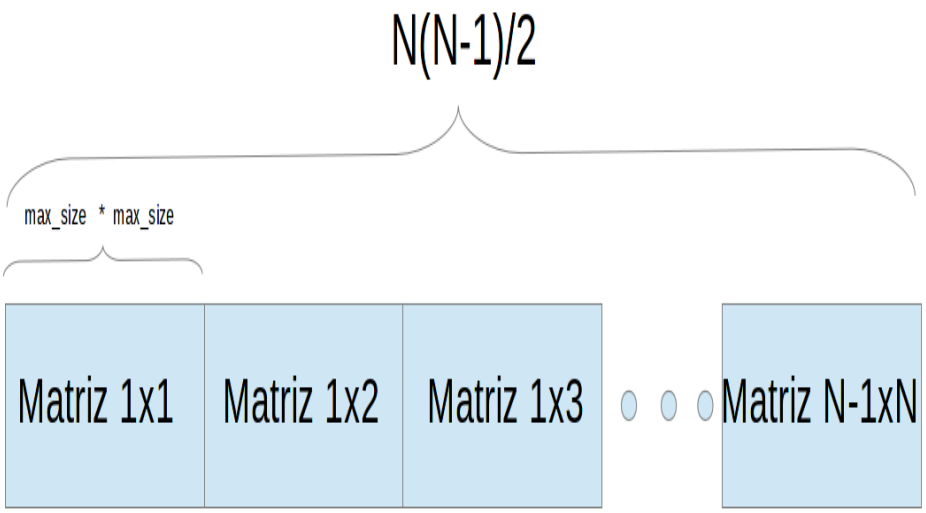

Figura 5.3: Estrutura do vetor de saída, onde *max_ size* é o maior tamanho de sequência e N é o número de sequências comparadas.

segunda diz que os dados do ponteiro referenciado no *buffer* devem ser copiados para essa memória. A Figura 5.3 ilustra o vetor de saída.

5.2.4 Criação do Programa *OpenCL* e Construção do *Kernel*

Após a definição dos objetos de memória, é carregado o arquivo contendo o código fonte do *kernel OpenCL*. A partir do código fonte do *kernel*, é construído um programa executável. Durante esse processo é possível criar *defines* para serem utilizados pelo *kernel*. Vale ressaltar ainda, que esse programa pode conter vários *kernels*. Para definir qual *kernel* deverá ser executado o nome do *kernel* é utilizado no método `cl::Kernel` que tem por parâmetros o executável do programa *OpenCL* e o nome do *kernel*. Obtendo sucesso na construção do *kernel*, são definidos seus parâmetros, que nesse caso são os vetores de sequência, tamanho e saída.

5.2.5 Definição de Dimensões, Execução e Coleta de Dados

Depois da geração do *kernel*, é preciso definir seu *index space*, ou seja, a quantidade de *work-groups* e *work-itens* que serão utilizados. Esses valores podem ser armazenados em variáveis do tipo `cl::NDRange`. Esse tipo de variável permite que sejam armazenadas informações sobre as 3 dimensões definidas pela *API OpenCL*, e que dependem das configurações do dispositivo escolhido. Tendo definidos esses valores, o *kernel* é executado utilizando o método `enqueueNDRangeKernel` através da variável de *command queue*. Podem ser utilizadas variáveis de evento para sincronismo com outras execuções ou utilizado para se medir o tempo de execução do *kernel*.

Assim que for terminada a execução do *kernel*, é executada a instrução *cl::finish()* que bloqueia a execução do código até que todas as *command-queues* sejam terminadas. Por

fim são lidos os dados da memória do dispositivo *OpenCL* para o *host* , utilizando o método `enqueueReadBuffer` da *command queue*, e liberados os objetos que foram utilizados durante a execução do programa.

Capítulo 6

Resultados Experimentais

Nesse capítulo serão mostrados os ambientes de testes utilizados bem como os resultados experimentais obtidos. Os dados coletados comparam os tempos de execução para tamanhos diferentes de sequências assim como a utilização de CPU durante a execução do programa.

6.1 Ambiente de testes

O ambiente de testes utilizado foi um notebook com 6GB de memória, 500GB de disco rígido e composto de CPU e GPU separadamente. A CPU utilizada foi uma Intel Core i7-2670QM, 4 *cores* com 2 threads cada e *clock* de 2.20GHz . A GPU utilizada foi uma Nvidia GeForce GT 525M de 1GB com 2 *Stream Multiprocessors* e 96 *Shading Units*. A Tabela 6.1 ilustra o ambiente de testes.

Além disso foi utilizado outro ambiente de testes, um computador *desktop* com 7.7GB de memória, disco rígido de 500GB e e composto de CPU e GPU separadamente. A CPU utilizada foi uma Intel core i7-2600k . As GPUS utilizadas foram duas placas de vídeo Nvidia GeForce GTX 580 com 1,5GB de memória com 16 *Stream Multiprocessors* e 512 *Shading Units*. Esse ambiente é ilustrado na Tabela 6.2.

O sistema operacional utilizado no primeiro ambiente foi a distribuição Linux *Arch Linux* com *kernel* x86_64 versão 3.6.9-1-ARCH. Para o reconhecimento dos dois dispositivos nesse sistema, foi necessário instalar tanto a *SDK* da Intel para OpenCL quanto a *NVIDIA GPU Computing SDK*, além dos *drivers* da placa de vídeo. No segundo ambiente foi utilizada a distribuição Linux Ubuntu 11.04 com *kernel* x86_64 versão 2.6.38-15-generic.

As sequências utilizadas na comparação foram criadas aleatoriamente através do programa mostrado no anexo B.1.

6.2 Tempos de Execução

6.2.1 Execução em GPU

Os tempos obtidos, em segundos, com a execução nas GPUs para se comparar 10 sequências de tamanho 10, 20, 40, 60, 80 e 100 resíduos são apresentados na figura 6.1 e 6.2. Como pode ser visto, os tempos de execução crescem de maneira exponencial com

Tabela 6.1: Primeiro ambiente de testes utilizado.

Device Type	CPU	GPU
Device Name	Intel(R) Core(TM) i7-2670QM CPU @ 2.20GHz	GeForce GT 525M
Vendor	Intel(R) Corporation	NVIDIA Corporation
Command Queue Properties	Out-of-order execution mode, Queue profiling	Out-of-order execution mode, Queue profiling
Address Bits	64	32
Execution Capabilities	Kernel Execution, Native Kernel Execution	Kernel Execution
Global Memory Cache Size	256 KB	32 KB
Memory Cache Type	Read Write	Read Write
Global Memory Cache Line Size	64 bytes	128 bytes
Global Memory Size	5,869 MB	1,024 MB
Host Unified Memory	Yes	No
Local Memory Size	32 KB	48 KB
Local Memory Type	Global	Local
Max Clock Frequency	2200	1200
Max Compute Units	8	2
Max Constant Buffer Size	128 KB	64 KB
Max Memory Allocation Size	1,468 MB	256 MB
Max Samplers	480	16
Max Workgroup Size	1024	1024
Max Work Item Dimensions	3	3
Max Work Item Sizes	(1024,1024,1024)	(1024,1024,64)
Max Write Image Arguments	480	8
Memory Base Address Alignment	1024	4096
Minimal Data Type Alignment Size	128 bytes	128 bytes
OpenCL C Version	OpenCL C 1.1	OpenCL C 1.1
Profile	FULL_PROFILE	FULL_PROFILE
Profiling Timer Resolution	1	1000
Vendor ID	OpenCL 1.1 (Build 31360.31426)	OpenCL 1.1 CUDA

Tabela 6.2: Segundo ambiente de testes utilizado.

Device Type	GPU
Device Name	GeForce GTX 580
Vendor	NVIDIA Corporation
Command Queue Properties	Out-of-order execution mode, Queue profiling
Address Bits	32
Execution Capabilities	Kernel Execution
Global Memory Cache Size	32 KB
Memory Cache Type	Read Write
Global Memory Cache Line Size	128 bytes
Global Memory Size	1,535 MB
Host Unified Memory	No
Local Memory Size	48 KB
Local Memory Type	Local
Max Clock Frequency	1544
Max Compute Units	16
Max Constant Buffer Size	64KB
Max Memory Allocation Size	256 MB
Max Samplers	16
Max Workgroup Size	1024
Max Work Item Dimensions	3
Max Work Item Sizes	(1024,1024,64)
Max Write Image Arguments	8
Memory Base Address Alignment	4096
Minimal Data Type Alignment Size	128 bytes
OpenCL C Version	OpenCL C 1.1
Profile	FULL_PROFILE
Profiling Timer Resolution	1000
Vendor ID	OpenCL 1.1 CUDA

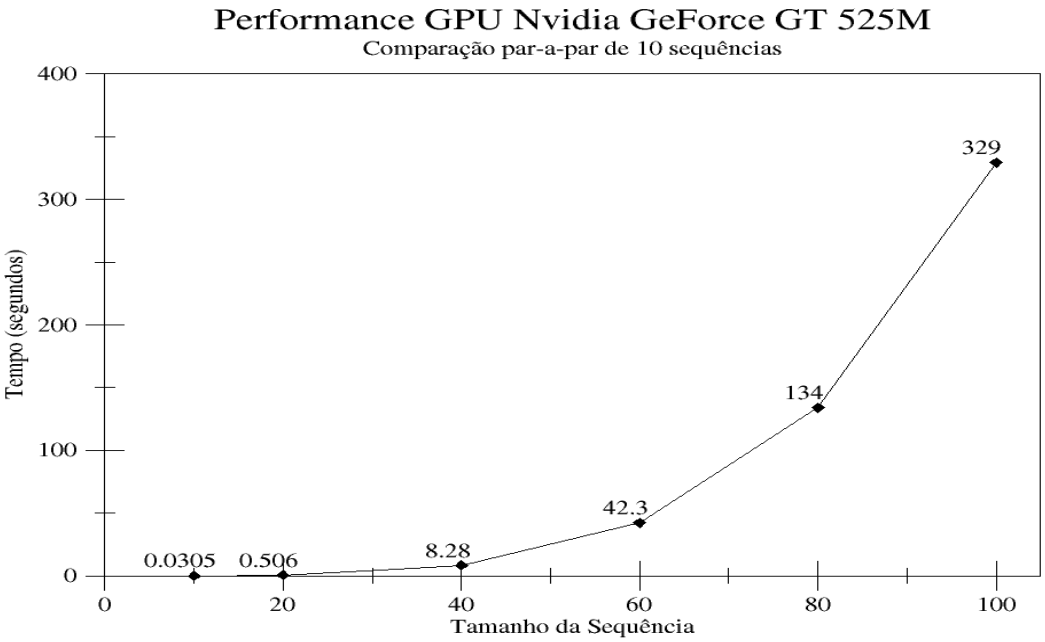

Figura 6.1: Tempo de execução do Kernel na GPU Nvidia GeForce GT 525M.

o aumento no tamanho da sequência. Isso ocorre porque um aumento de 20 resíduos no tamanho da sequência corresponde ao aumento de 20 resíduos em 45 comparações.

6.2.2 Execução em CPU

Para a execução em CPU, foram também feitas 45 comparações (10 pares de sequências) de tamanho 10, 20, 40, 60, 80 e 100. Como pode ser visto, o crescimento no tamanho da sequência também implica em um crescimento exponencial no tempo de execução.

6.2.3 Comparação entre GPU e CPU

Como pode ser visto na Figura 6.4, a execução em CPU é, surpreendentemente, muito mais rápida do que a execução em GPU. Isso pode ter ocorrido devido a diversos fatores. Primeiramente, confirmamos que a GPU realmente está sendo usada nas comparações porém não foi possível recuperar quantos *cores* da GPU foram realmente utilizados. Caso poucos *cores* tenham sido utilizados, isso explicaria o desempenho baixo da GPU. Em segundo lugar, a GPU utilizada nos testes é bastante simples, com clock bem mais baixo que a CPU e menos recursos de memória do que o Intel Core i7 6.1. Isso também poderia explicar o baixo desempenho da GPU, quando comparado à CPU.

6.2.4 Monitoramento dos recursos do sistema

Durante a execução do programa, foram monitorados os recursos de sistema. Em relação a memória utilizada, para 10 sequências de tamanho 100 foram utilizados quase 4MB

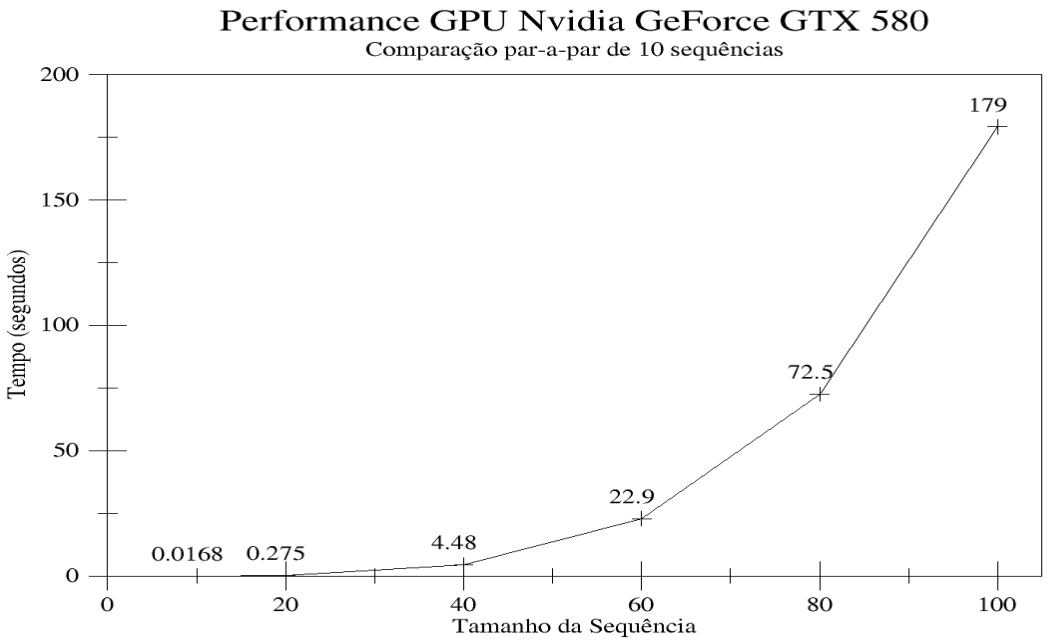

Figura 6.2: Tempo de execução do Kernel na GPU Nvidia GeForce GTX 580.

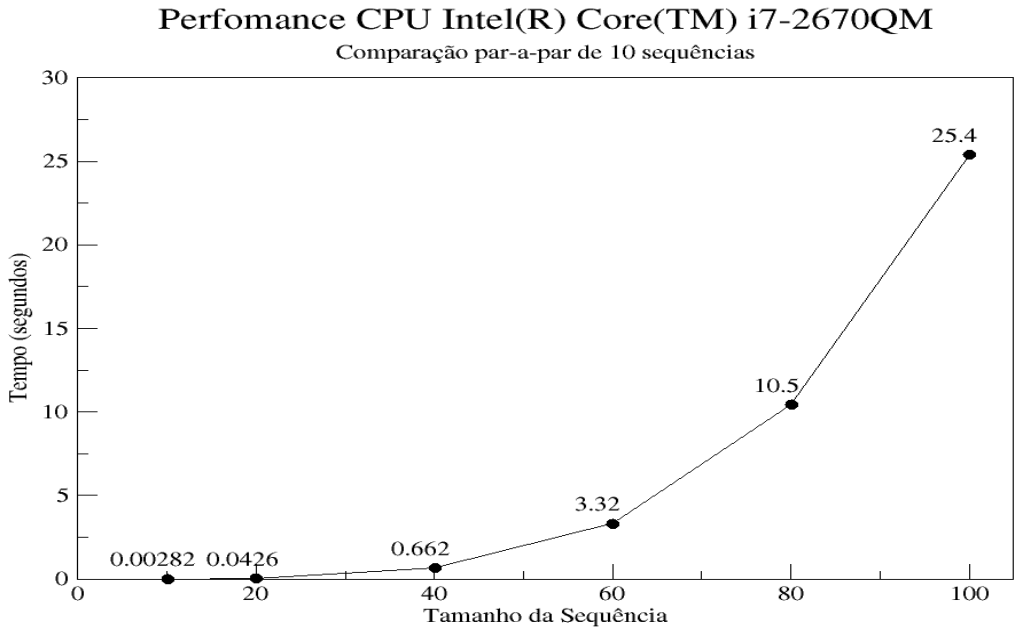

Figura 6.3: Tempo de execução do Kernel na CPU.

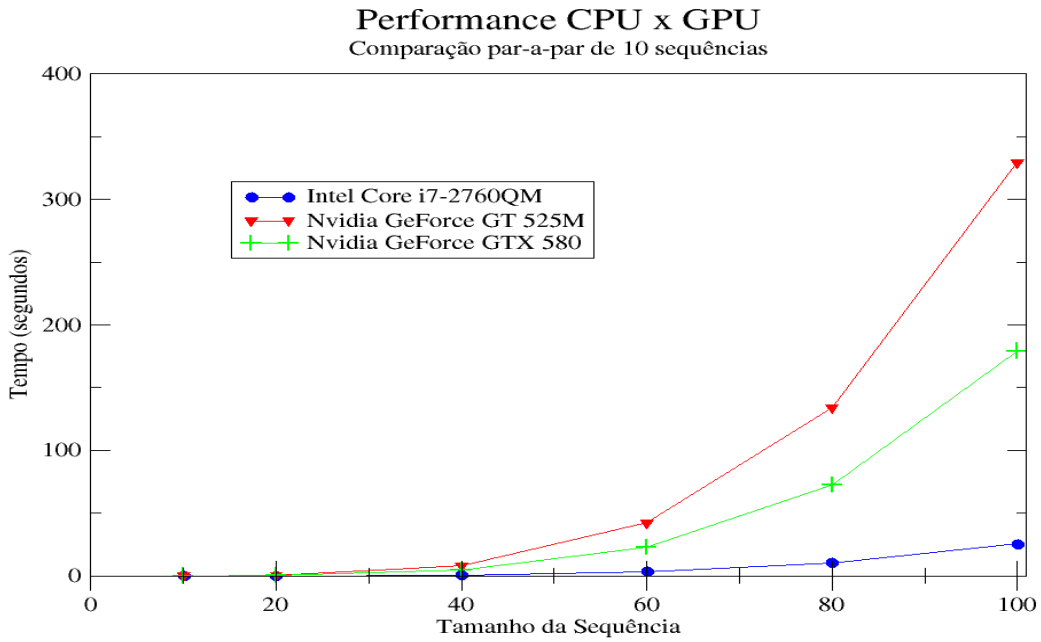

Figura 6.4: Comparação de performance da CPU e GPU.

o que não é um fator limitante tanto para as GPUs quanto para as CPUs. Assim, devido ao pouco espaço utilizado em memória, não foram gerados gráficos de consumo de memória. Quanto a utilização de CPU, foi percebido um comportamento não esperado como é possível ver na figura 6.5. Durante todo o processamento do programa em CPU, a utilização de CPU fica centralizada em apenas 1 *core* ficando em 100% em praticamente toda a execução enquanto o restante dos *cores* não chegam a ultrapassar 5% de consumo,em média. Como o ambiente utilizado possuía 8 *cores* (4 reais e 4 virtualizados) esperava-se a distribuição do processamento entre eles. Isso levanta dúvidas se a implementação do OpenCL para processadores Intel está preparada para utilizar todos os núcleos ou se o falta alguma configuração prévia para utilizá-los.

Durante a execução na placa de vídeo observou-se um comportamento semelhante, como mostra a figura 6.6. A diferença foi que, no início da execução, um *core* fica em torno de 40% de consumo de CPU e após algum tempo há uma queda no consumo desse *core* e a elevação do consumo de um outro *core*. Esse último mantém o consumo em torno de 40% até a finalização da execução, enquanto o restante dos *cores* não passam de 5%.

Apesar dos esforços, não foi possível obter informações do processamento da GPU, pois as ferramentas de *profilling* testadas apresentaram problemas diversos, tanto na instalação quanto durante a execução, que impossibilitaram o uso delas.

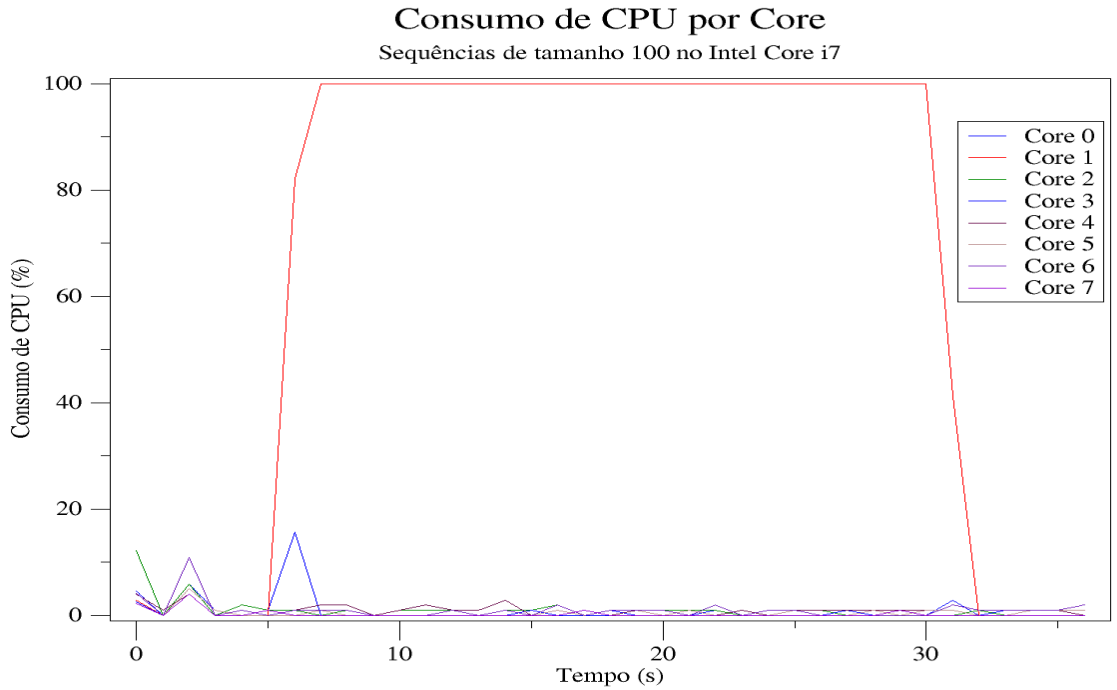

Figura 6.5: Consumo CPU por *core* no processamento de 10 sequências de tamanho 100 utilizando a CPU.

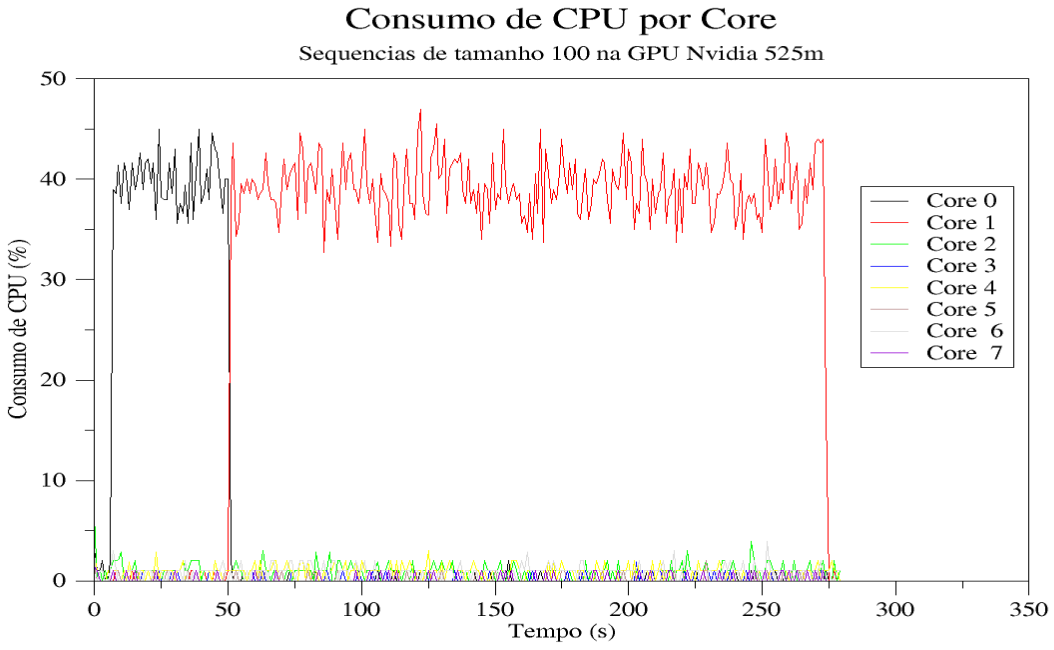

Figura 6.6: Consumo CPU por *core* no processamento de 10 sequências de tamanho 100 utilizando a GPU.

Capítulo 7

Conclusão

O presente trabalho de graduação propôs e a analisou uma implementação do algoritmo de Smith-Waterman em OpenCL com o objetivo de avaliar a adequação da estratégia para plataformas heterogêneas. No desenvolvimento desse projeto, optou-se por utilizar um único *kernel* para fazer a comparação par-a-par entre sequências, divididas em *work-groups*. Nos ambientes disponíveis foi possível utilizar tanto a CPU quanto a GPU na realização dos testes.

Os resultados obtidos em duas GPUs e em um *multicore* mostram que é possível a programação para multi-plataformas, com praticamente nenhuma alteração no código, utilizando OpenCL de forma simples, porém não foi possível comprovar a utilização de 100% do potencial dos dispositivos envolvidos. O projeto utilizado obteve performance muito melhor em *multicore* quando comparado à performance nas duas GPUs. Supomos que a baixa performance em GPU pode ter sido causada pela não utilização de todo o potencial das GPUs e que provavelmente não foram utilizados todos os cores. Esse problema pode ter ocorrido seja por parte da implementação do *kernel* seja pelo nível de maturidade da implementação do OpenCL para esse tipo de dispositivo e/ou para o sistema operacional escolhido.

Como trabalhos futuros sugere-se primeiramente gerar o alinhamento das sequências e também o refinamento do *kernel* utilizado, visto que o projeto não teve por foco obter a melhor performance possível. A utilização de múltiplos dispositivos por contexto, também é uma sugestão, visto que aumentando o paralelismo pode-se diminuir o tempo de execução, possibilitando aumentar a quantidade e o tamanho das sequências. Outro ponto a ser verificado é o melhoramento do *kernel* para a comparação de duas sequências, de modo a aceitar sequências mais longas e dividir o processamento entre múltiplos *work-groups* e analisar a possibilidade de ser feito o mesmo para o caso de várias sequências e múltiplos dispositivos. Outra sugestão seria a criação de uma interface gráfica para facilitar tanto as entradas e saídas de dados, bem como a seleção de dispositivos a serem utilizados.

Referências

[1] AMD. OpenCL: The Open Standard for Parallel Programming of GPUs and Multicore CPUs. http://www.amd.com/us/products/technologies/stream-technology/opencl/pages/opencl.aspx. [Online; Acessado em 20 de Outubro de 2011]. 19

[2] J. Backus. Can programming be liberated from the von neumann style?: a functional style and its algebra of programs. *Commun. ACM*, 21:613–641, Agosto 1978. 12

[3] A. Boukerche, A.C.M.A. de Melo, M. Ayala-Rincón, and T.M. Santana. Parallel smith-waterman algorithm for local dna comparison in a cluster of workstations. In *Proceedings of the 4th international conference on Experimental and Efficient Algorithms*, WEA'05, pages 464–475, Berlin, Heidelberg, 2005. Springer-Verlag. 1

[4] A.R. Brodtkorb, C. Dyken, T.R. Hagen, J.M. Hjelmervik, and O.O. Storaasli. State-of-the-art in heterogeneous computing. *Scientific Programming*, 18:1–33, Janeiro 2010. 11, 12

[5] Intel Corporation. Sandy Bridge: The 2nd Generation Intel Core Processor is here. http://getsmart.intel.com/uk/technology/single-view/article/sandy-bridge-the-2nd-generation-intelR-core-processor-is-here/. [Online; Acessado em 20 de Novembro de 2011]. ix, 13

[6] F.H.C. Crick and J.D. Watson. A structure for deoxyribose nucleic acid. *Nature*, 171:737–738, 1953. 3

[7] E.F. de O Sandes and A.C.M.A. de Melo. Smith-waterman alignment of huge sequences with gpu in linear space. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 1199 –1211, may 2011. 1

[8] E.V. DeNardo. *Dynamic Programming: Models and Applications*. Dover Books on Mathematics. Dover Publications, 1982. 1, 6

[9] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological sequence analysis: probabilistic models of proteins and nucleic acids*. Cambridge University Press, 1998. 4

[10] M.J. Flynn. Some computer organizations and their effectiveness. *IEEE Trans. Comput.*, 21:948–960, September 1972. 13

[11] Society for Science & the Public. http://www.sciencenews.org/pictures/021211/alphabet_code_chart_zoom.gif. [Online; Acessado em 16 de Outubro de 2011]. xi, 5

[12] Khronos Group. http://www.khronos.org/assets/uploads/developers/library/overview/OpenCL-Overview-Jun10.pdf. [Online; Acessado em 20 de Outubro de 2011]. ix, 1, 19, 21, 25

[13] J.L. Hennessy and D.A. Patterson. *Computer Architecture, : A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, fourth edition, 2006. 13

[14] Apple Inc. Opencl Programming Guide for Mac OS X. http://developer.apple.com/library/mac/documentation/Performance/Conceptual/OpenCL_MacProgGuide/OpenCL_MacProgGuide.pdf. [Online; Acessado em 20 de Novembro de 2011]. 20

[15] A. Ly. OpenCL Course: Introduction to OpenCL Programming. http://developer.amd.com/zones/OpenCLZone/courses/Documents/Introduction_to_OpenCL_Programming%20(201005).pdf, Maio 2010. [Online; Acessado em 04 de Outubro de 2011]. ix, 11, 28

[16] D.W. Mount. *Bioinformatics: sequence and genome analysis*. Cold Spring Harbor Laboratory Press, 2004. ix, 1, 3, 4, 6

[17] A. Munshi, B. Gaster, T.G. Mattson, J. Fung, and D. Ginsburg. *OpenCL Programming Guide*. OpenGL Series. Pearson Education, Limited, 2011. ix, 20, 22, 26, 27

[18] S.B. Needleman and C.D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443 – 453, 1970. 7

[19] Net Industries. History of the Evolution of the Components of the Video card. http://encyclopedia.jrank.org/articles/pages/212966/Video-card.html, 2012. [Online; acessado em 10 de Janeiro de 2013]. 14

[20] M.S. Nicholas. Remote Sensoring Tutorial. http://rst.gsfc.nasa.gov/Sect20/dna_versus_rna_reversed.jpg. [Online; Acessado em 16 de Outubro de 2011],. ix, 4

[21] NVIDIA Corporation. NVIDIA CUDA C programming guide, 2012. Version 4.2. ix, 15, 16

[22] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yelick. A Case for Intelligent RAM. *IEEE Micro*, 17:34–44, March 1997. 12

[23] The Linux Documentation Project. The Linux Gamers' HOWTO. http://tldp.org/HOWTO/Linux-Gamers-HOWTO/x608.htm, 2012. [Online; acessado em 10 de Janeiro de 2013]. 14

[24] A. Silverstein, V.B. Silverstein, V. Silverstein, and L.S. Nunn. *DNA*. Science Concepts. Second Series. Lerner Pub Group, 2008. 3

[25] AMD Staff. Opencl and the AMD APP SDK. http://developer.amd.com/documentation/articles/PublishingImages/opencl_figure3.jpg. [Online; Acessado em 29 de Outubro de 2011]. ix, 23

[26] J.E. Stone, D. Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in Science Engineering*, 12(3):66–73, May-June. 11

[27] Khronos OpenCL Working Group. The OpenCL Specification. http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf. [Online; Acessado em 02 de Outubro de 2011]. 24, 25, 27, 29, 32

[28] Khronos OpenCL Working Group. The The OpenCL C++ Wrapper API. http://www.khronos.org/registry/cl/specs/opencl-cplusplus-1.1.pdf. [Online; Acessado em 18 de Outubro de 2011]. 34

[29] M.S. Waterman and T.F. Smith. Identification of common molecular subsequences. *J. Mol. Biol.*, 147:195–197, 1981. 1, 7, 8, 9

[30] J. Xiong. *Essential bioinformatics*. Cambridge University Press, 2006. 4, 5

Apêndice A

Código Exemplo de um programa utilizando OpenCL

```
1  #include <CL/cl.hpp>
2  #include <string>
3  #include <fstream>
4  #include <vector>
5  #include <iomanip>
6  #include <unistd.h>
7
8  const int ELEMENTS = 2048; // numero de elementos em cada vetor
9
10 int main (int argc, char* argv[])
11 {
12 // Consulta Plataformas Disponiveis
13 std::vector<cl::Platform> platforms;
14 cl::Platform::get(&platforms);
15 if (platforms.size() == 0) {
16   std::cout << "Platform␣size␣0\n";
17   return -1;
18 }
19 // prepara as variavei de entrada e saida
20 int *A = new int[ELEMENTS]; // arrays de entrada
21 int *B = new int[ELEMENTS];
22 int *C = new int[ELEMENTS]; // Output array de saida
23
24 //Inicializa arrays de entrada
25 for(int i = 0; i < ELEMENTS; i++) {
26       A[i] = i;
27       B[i] = i;
28    }
29
30 // Prepara propriedades do contexto
31 cl_context_properties properties[] =
32 { CL_CONTEXT_PLATFORM, (cl_context_properties)(platforms[0])(), 0};
33
34 // cria contexto agrupando dispositivos do tipo CPU
35 cl::Context context(CL_DEVICE_TYPE_CPU, properties);
36
37 //carrega dispositivos em um vetor
```

```
38 std::vector<cl::Device> devices = context.getInfo<CL_CONTEXT_DEVICES>();
39
40 cl::Buffer d_A, d_B; // Input buffers on device
41 cl::Buffer d_C; // Output buffer on device
42 cl_mem_flags buffer_options = CL_MEM_READ_ONLY | CL_MEM_ALLOC_HOST_PTR |
       CL_MEM_COPY_HOST_PTR ;
43
44 d_A = cl::Buffer(context, buffer_options , ELEMENTS * sizeof(int),(void *)A);
45 d_B = cl::Buffer(context, buffer_options , ELEMENTS * sizeof(int),(void *)B);
46 d_C = cl::Buffer(context, CL_MEM_READ_WRITE, ELEMENTS * sizeof(int),(void *)C);
47
48 //Essa funcao carrega o codigo fonte do programa
49 cl::Program::Sources *source;
50 const char *sourceFilename = "vector_add.cl";
51 std::ifstream sourceFile(sourceFilename);
52 std::string sourceCode(std::istreambuf_iterator<char>(sourceFile), (std::
       istreambuf_iterator<char>()));
53
54 // cria source para o kernel
55 source = new cl::Program::Sources(1, std::make_pair(sourceCode.c_str(), sourceCode.
       length()+1));
56 sourceFile.close();
57
58 // cria o programa a partir do source
59 cl::Program program_ = cl::Program(context, *source);
60
61 // constroi o programa para os dispositivos no vetor
62 program_.build(devices);
63
64 // cria o kernel
65 const char *kernel_name = "vector_add";
66 cl::Kernel kernel(program_, kernel_name);
67
68 cl::Event event;
69 // cria a CommandQueue
70 cl::CommandQueue queue(context, devices[0], 0);
71
72 //executa o kernel
73 queue.enqueueNDRangeKernel( kernel, cl::NullRange, cl::NDRange(4,4), cl::NullRange,
        NULL, &event);
74
75 // Aguarda o evento de termino da execucao para ler os dados de saida
76 event.wait();
77 queue.enqueueReadBuffer(d_C, CL_TRUE, 0,(size_t) ELEMENTS*sizeof(int), C);
78 }
```

Listagem A.1: Exemplo de código para um programa OpenCL para adição de matrizes utilizando o kernel mostrado na Listagem 4.2

Apêndice B

Gerador de sequências aleatórias

```
#include <fstream>
#include <random>
#include <stdlib.h>
//para GCC <4.7 -std=c++0x ou -std=gnu0x para compilar
//para GCC >=4.7 -std=c++11 ou -std=gnu++11 para compilar

int irand(int min, int max) {
    return ((double)rand() / ((double)RAND_MAX + 1.0)) * (max - min + 1) + min;
}

int main (int argc, char* argv[]){

  int N_FILES = 10;
  int SIZE = 100;
  char A[4];

  int index =0;
  char filename[40];
  std::ofstream myfile;

  A[0]='A';
  A[1]='T';
  A[2]='C';
  A[3]='G';

  srand(time(0));
  if(argc==3){
    N_FILES = atoi(argv[1]);
    SIZE = atoi(argv[2]);
  }
  for(int k=0; k<N_FILES;k++){

    sprintf(filename, ".seq/sequence%d.txt", k);

    myfile.open (filename,std::ios_base::trunc);
    for(int i=0; i< SIZE;i++){
      index = irand(0,3);
      myfile << A[index] ;
    }
    myfile.close();
```

```
41   }
42   return 0;
43 }
```

Listagem B.1: Código para gerar sequências aleatórias